Conductor Hero

Yuta Kuboyama, Natalie Cheung, and Edgar Twigg
6.111 Final Project Report
December 14, 2007

Abstract

Recent times have seen many games based on music in which a user is challenged to perform (dancing, simulated guitar playing, etc) a task in time with a given score. Conductor Hero flips this paradigm, allowing the user to command the game to perform a given score at whatever speed and in whatever style the user desires. The user stands in front of a camera and waves a baton. Conductor hero gathers information on how the user wishes the score to be performed based on the movement of this baton, and plays the score accordingly.

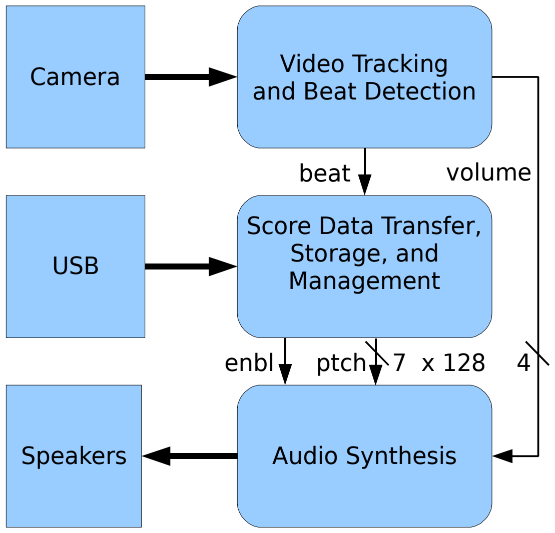
Table of Contents

List of Figures

Overview
Sitting in the audience and listening to the conductor conduct his orchestra is an event that many of us have experienced. However, being the conductor is a rarity. Conductor Hero allows anyone to be the conductor of an orchestra. This project takes simple hand movements from the conductor and outputs the sound of an orchestra playing in the tempo and style of the conductor's motion. Furthermore, as in real life, the conductor chooses which piece he wants to play.

If the user wishes to play any song, all he needs to do is acquire a MIDI file with the desired music. The system can automatically parse the file, convert it to the conductor-hero format, and send it to the labkit. Once on the labkit, the system plays through the score at the tempo he conducts, and at the volume he conducts!

Description
Our project is composed of three parts: the conductor interface, score management, and audio synthesis. The conductor module uses a video camera which tracks the motion of a colored ball. As the user conducts, circuitry extracts dynamics and tempo from the ball's motion. The beats that the conductor makes are be sent to score management and the dynamics are sent to the audio synthesis module. The score management system then reads through the score data in the tempo conducted by the conductor. As the score management system reads through the score, it commands the audio synthesis unit to enable certain instruments at certain pitches. Based on these enable and pitch signals form score management and dynamic signals from the conductor, the audio synthesis unit outputs appropriate audio signals to the speakers.


Camera
Video Tracking and Beat Detection
beat
volume
USB
Score Data Transfer, Storage, and Management
enbl
ptch
7
x 128
4
Speakers
Audio Synthesis
Figure 0. System Interfaces

Video Tracking and Beat Detection (Natalie Cheung)
The conductor module consists of two parts: the video decoder module and the object detector module. The video camera data is sent to the FPGA where it is stored and read. After the video decoder module is finished, the object detector module detects the object and outputs the center of mass coordinates of the object as a crosshair on the computer screen. This output as shown in Figure 1 is what the user sees and uses to conduct the orchestra.

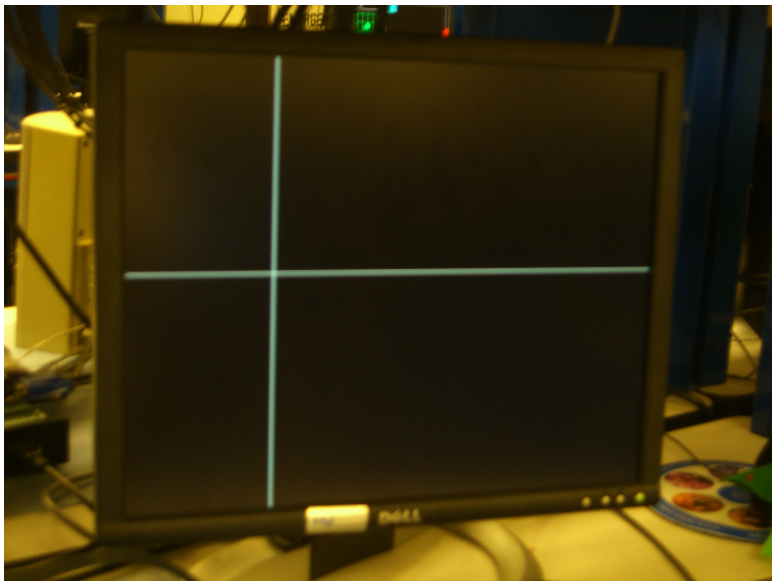


Figure 1. Video Output: The crosshairs on the computer screen show the location of the object.

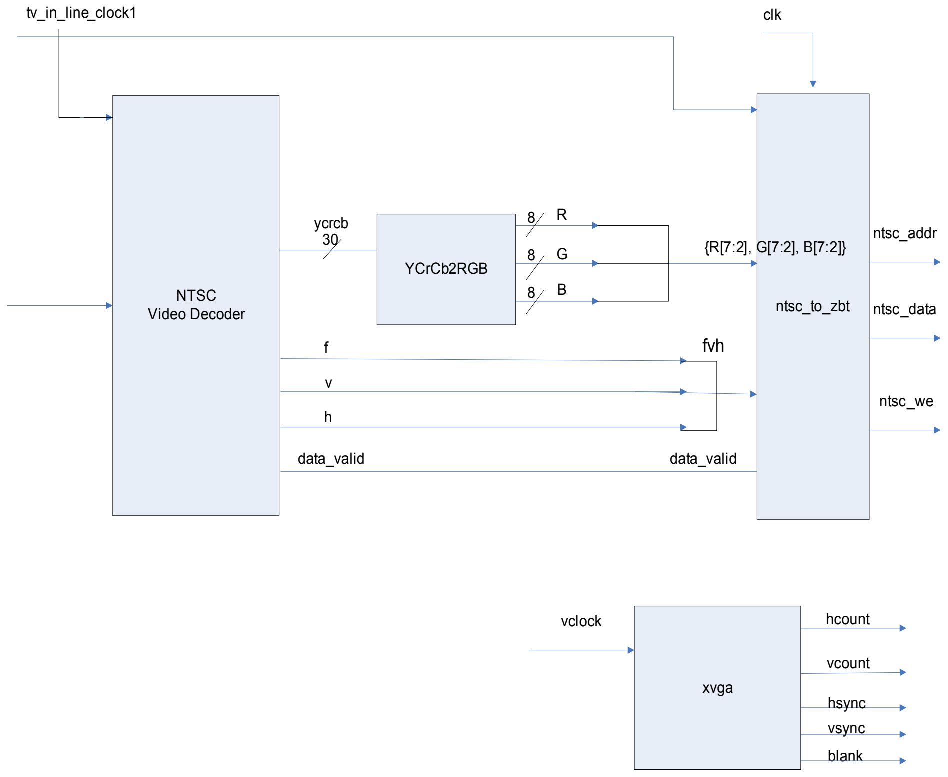


Figure 2. Video Decoder Modules

Video Tracking and Beat Detection: NTSC decode, ADV7185 Video Decode
These modules were provided on the 6.111 website. The ADV7185 Video Decoder Module uses the I2C interface to decode the analog NTSC signal from the video camera and output a stream of digital LLC data. The stream of digital data is then passed onto the NTSC decode module which was written by Javier Castro. The NTSC/PAL video decoder generates hsync, vsync, and field signals and 30 bit pixels in the YCrCb format. These pixels show the luma and blue and red chroma in a picture. This module was not modified for the project.

Video Tracking and Beat Detection: YCrCb2RGB
This module takes the luminance and chrominance found in YCrCb and converts it into 8 bits of red, green, and blue signals. It is important to convert from YCrCb to RGB before putting it into the ZBT because RGB reduces storage and bandwidth. Furthermore, it is necessary to change the signal to RGB so that the computer monitor outputs a color screen for the user. This module was found in the Xilinx application notes. This module was not modified for the project.

It is useful to note that converting from YCrCb to RGB was best for the project. A HSV (Hue, Saturation, Value) module was tested to see if it could detect the baton better than the RGB module. The module consisted of converting the red, green, and blue signals into an eight bit value for hue. Although it might have been easier to detect with hue, it was not necessary to do so because the threshold_finder module detected the baton accurately and precisely. Furthermore, hue could not be accurately displayed on the computer monitor which would cause some trouble during testing.

Video Tracking and Beat Detection: NTSC_to_ZBT
The ntsc_to_zbt module takes the decoded signals from the NTSC decoder and the output of the YCrCb2RGB module and stores it into the ZBT ram. Because the memory is 36 bits wide, it is efficient to take in 18 bits of video data so that each address contains two 18 bits of RGB. The module, which was written by I. Chuang, was tweaked so that the data in took in 18 bits (6 bits each of red, green, and blue) and would store 18 bits worth of data in an address.

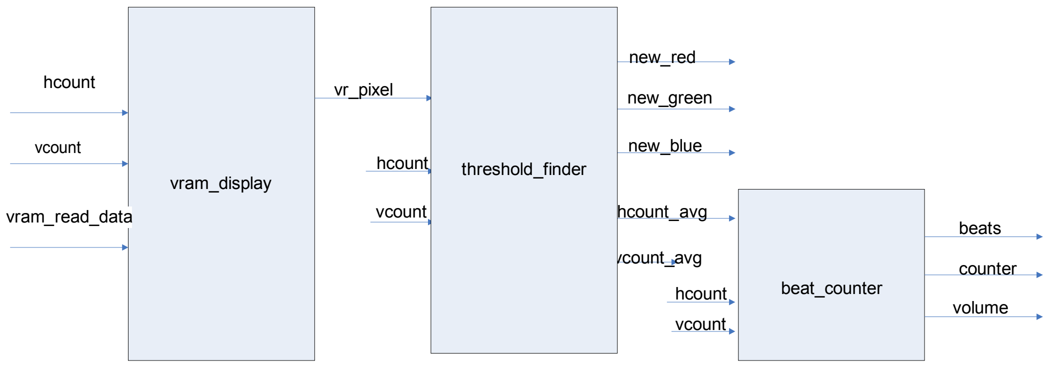


Figure 3. Object Detection Modules

Video Tracking and Beat Detection: vram_display
The data from the ZBT ram is read in the vram_display module. Because the ZBT ram has two cycles of read and write latency, the module latches the data. The module grabs the information from the ZBT and outputs the pixel data. This module was given to us on the 6.111 website, but was changed so that the vram_display would output an 18 bit signal to be displayed on the monitor.

Video Tracking and Beat Detection: Threshold_finder
After testing that the monitor could output color from the video camera, the threshold_finder module separates the ball/baton from the rest of the video data sent to the screen. The goal of the module was to detect a certain range for the pixels and output the new signals and the center of mass of the ball to the screen. At every clock cycle, the module looks at the red, green, and blue signals sent to the module and checks to see if it outputs the color of the ball (bright pink). If these signals pass the threshold test, the module outputs the red, green, and blue signals of the certain pixel. Furthermore, to find the center of mass, it adds all the x coordinates of the baton and at the end of each frame, divides the summation of x coordinates by the number of pixels that have passed the threshold. The module also finds the y coordinate of the baton in a similar manner. To get rid of speckles that were found on the monitor, the module makes sure that the divisor (the number of pixels in that frame) is greater than 30. If the divisor is less than thirty, then the center of mass coordinates do not change. Inside the threshold_finder module, there are two instantiations of a divider module that was created by the Coregen found in the Xilinx program. These two instantiations are used to find the center of mass coordinates for the baton.

Video Tracking and Beat Detection: RGB_value_finder
In order to find the precise RGB threshold values, the RGB_value_finder was created. This module allows the user to use the FPGA labkit buttons to detect what the red, green, and blue values are at a certain pixel. When the hcount and vcount are at the specific location, the red, green, and blue values are outputted. However, if the hcount and vcount are not at the location, the red, green, and blue values stay the same. The RGB values are shown on the 16 hex display so that one can find the maximum and minimum red, green, and blue values that the threshold_finder module is suppose to detect.

Video Tracking and Beat Detection: beat_counter

Using the center of mass coordinates of the baton, the beat_counter can detect the beat and volume that the user is conducting. The beat output is a square wave, each change from low to high or high to low denotes a beat. The volume is a four bit output where 4'b0 is *ppp* (very soft) and 4'b1111 is *fff* (extremely loud). At the end of each frame, the x coordinate of the center of mass is put into a register. There are seven registers, each remembering the register value of the register before them. Consequently, at each frame, the module knows the center of mass position of the baton seven frames before. The module only needed to take in the x coordinate of the center of mass because the every movement, even up and down movements, had a slight change in the x coordinates. Thus, only using the x coordinates was sufficient.

In order to make sure that the user is ready to start a beat, the module waits for 180 frames (about three seconds) before it processes any information. This was done by comparing the current position of the baton to the position seven frames earlier. If the position was within a certain range on the screen, plus or minus five pixels, then the counter would begin counting. A range of ten pixels was used because it was tested that this was the maximum range that the user could hold the baton at a comfortable position. If the center of mass position ever changed rapidly, the counter would be reset. When the counter reaches 180 frames, the LED turns on to signify that the baton has been calibrated and is ready to detect beats.

Once the counter reaches the 170th frame, the x coordinate of the center of mass is put into a register. This is used to figure out if the user would move the baton left or right on his first movement. If the object is moved to the right, the module realizes that the user is moving to the east. Similarly, the module knows that moving the baton left is the same as moving to the west. Once a direction has been created, the module waits until one of the following happens: a glitch in the beat or a change in direction. If the location of the center of mass goes in the opposite direction, the module checks if the difference in the present and previous locations is greater than five pixels. If so, then it realizes that the user has switched directions and thus has made a beat. Consequently, the beat signal always switches from high to low or low to high.

The volume, the absolute value of the distance between the current location and the previous location is calculated. Because the difference between the two beat locations is greater than four bits, the module bit shifts the difference so that volume becomes a four bit number. One problem that arose was that the volume would change from a *forte* (loud) to a *piano* (soft) too quickly. This was fixed by using the volume_control module.

Video Tracking and Beat Detection: volume_control

The volume_control module is used to filter the noisy volume to a clean volume. Because the volume would occasionally change more than two bits per frame, this caused the music to sound jumpy and unprofessional. The clean volume allows the volume to decrement or increment by one so that the transition from loud to soft would be smooth. Consequently, at the end of every fifteen frames, the volume_control module would compare the noisy and clean volumes and increment the volume if the noisy volume was greater than the clean volume or decrement the volume if the noisy volume was lower than the clean volume.

Score Data Transfer, Storage, and Playback (Edgar Twigg)

Conductor Hero's score system has four main components. The first of these is converting MIDI files into a format suitable for Conductor Hero. The second is providing a mechanism to store the entire score in memory at once (we used Flash ROM for this purpose). The third capability is transferring converted MIDI files to the labkits memory. And finally, with the foundation supplied by capabilities one to three, the system needs the capability to read the score back in sync with the beat signal supplied by the conductor, and generate the appropriate enable and pitch signals to send to the audio synthesis unit. We present these four capabilities in that order.

Score Data Transfer, Storage, and Playback: Data Format and MIDI Conversion: Time Scheme
To enable scores to be played back sensibly at any tempo, time is counted in terms of beats and ticks rather than seconds. Each beat is divided into 128 ticks. At a very slow 50 bpm, 128 ticks per beat allows for a time resolution of 9 ms, which is safely less a person's ability to distinguish time interval differences (approximately 30 ms). A problem we encountered with this time scheme is that many MIDI files divide beats into 192 divisions. As a result, fast rhythms are aliased when converted to 128 divisions per beat, and this aliasing is just barely audible.

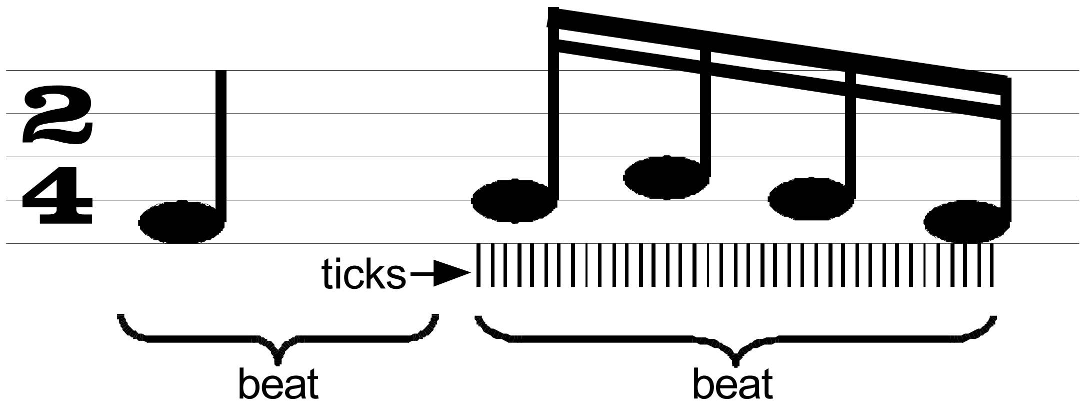


Figure 4. Time Scheme

Score Data Transfer, Storage, and Playback: Data Format and MIDI Conversion: Note Data Format
Score data is stored in Flash ROM in the Note Data format. The first 12 bits specify what beat the note is in. The second 7 bits specify what tick the note starts on. The third 7 bits specify which instrument should play the note. The fourth 7 bits specify the pitch the note should play. And the very last bit specifies whether the note should be turned on or off. These bit assignments were carefully tuned. We can address 128 instruments and command them at 128 different pitches (significantly more than a piano's 88 keys).

Start Beat	Start Tick	Instrument	Pitch	Enable	Unused
47-36 (0-4095)	**35-29** (0-127)	**28-22** (0-127)	**21-15** (0-127)	**14** (0-1)	**13-0**

Figure 5. NoteData Bit Allocation

Score Data Transfer, Storage, and Playback: Data Format and MIDI Conversion: Software
Parsing MIDI files turned out to be much harder than originally expected. MIDI holds data in packets of variable length, and a lot of bit-level manipulation is required to get at the data contained within. To simplify this process, we used the Rogus McBogus library created by Ben Denckla and Patrick Pelletier at the MIT Media Laboratory. We used this to parse the MIDI file and put its contents into a more easily accessible format.

Next, we go through every event in the MIDI file and extract the information necessary to make a NoteData event. Once we have gone through the entire MIDI file, we sort the extracted NoteData events in chronological order, and write them to a human readable Conductor Hero Text file (*.cht).

```
BEAT  TICK  INST  PTCH  ENBL
4095  127   127   127   1
4     0     12    46    1
4     0     9     82    1
4     0     11    62    1
4     0     15    57    1
4     0     6     58    1
4     0     1     82    1
4     0     10    70    1
4     0     13    34    1
4     0     14    46    1
4     6     9     82    0
```

Figure 6. cht file format example (beginning of Star Wars in this case)

The .cht file is exactly what we want to put in labkit memory, only it is in text form rather than binary. To convert it to binary, we go through the file line by line and convert it into a bitstream, which can be sent byte by byte to the labkit.

Score Data Transfer, Storage, and Playback: Flash ROM
Flash ROM is a HUGE hassle. Initially, we tried to write a module that interfaced with the Flash ROM using the the Flash ROM datasheet as a reference. After twenty hours with no results, we decided to instead modify the test code written by Nathan Ickes in 2005. We HIGHLY recommend Nathan's code as a starting point for anyone wishing to include Flash ROM in their project.

There are three modules which are responsible for interfacing with the ROM: flash_manager, test_fsm, and flash_int.

Score Data Transfer, Storage, and Playback: Flash ROM: flash_int
The flash_int module was written by Nathan Ickes and was totally unmodified for our project. This module manipulates the write enable, clear enable, and output enable of the flash to abstract the process of writing a word to an address.

Score Data Transfer, Storage, and Playback: Flash ROM: test_fsm
This module was originally written by Nathan Ickes and was heavily modified for use in our project. Although flash_int allows you to manipulate flash at a very low level, working with flash requires a sequence of reads and writes to produce one actual read or write. This module carries out those sequences. It is capable of four actions:

Write Initialize - unlocks the flash for writing and clears the memory
Write - writes one word to the ROM
Read Initialize – puts the flash into read mode
Read – reads one word from ROM

Score Data Transfer, Storage, and Playback: Flash ROM: flash_manager
This module allows the programmer to interact with flash almost as though it were a BRAM with a busy signal. It's only restriction is that memory must be written sequentially, but this is not a major restriction since it interfaces with a ROM and is thus unsuitable for anything where flexible write capabilities are required anyway.

If writemode is active and reset goes high, the flash_manager commands test_fsm to wipe its memory and asserts that it is busy until test_fsm finishes. Once busy goes low, flash_manager is ready to write. If write is asserted, flash_manager stores the data on its input data pins and asks the test_fsm to go through the appropriate sequence to write this data to ROM.

If writemode is inactive and read is asserted, the flash_manager asks the test_fsm to read the given address. If the flash is already in read mode, test_fsm simply reads off the address. If it isn't already in read mode, it first activates read mode, and then reads off the address.

Score Data Transfer, Storage, and Playback: Write Mode Operation
While in write mode, the entire playback system is held in reset while the Flash ROM is loaded with data over USB. An FTDI UM245 unit deals with the USB protocol and buffers incoming data in a FIFO queue. The usb_input module manipulates the FTDI chip's inputs and reads its outputs to read the data byte by byte and present it to the data_accum module. The data_accum module strings bytes together into 16 bit words which are then passed to the Flash ROM.

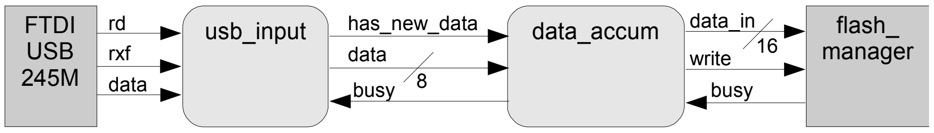


Figure 7. Block diagram of modules involved in writing

Performance wise, a byte of data can be read from USB in 588 ns. To write two bytes of data takes 154.4 us (hundreds of times longer). Thus, the performance bottleneck during writing is by far Flash ROM.

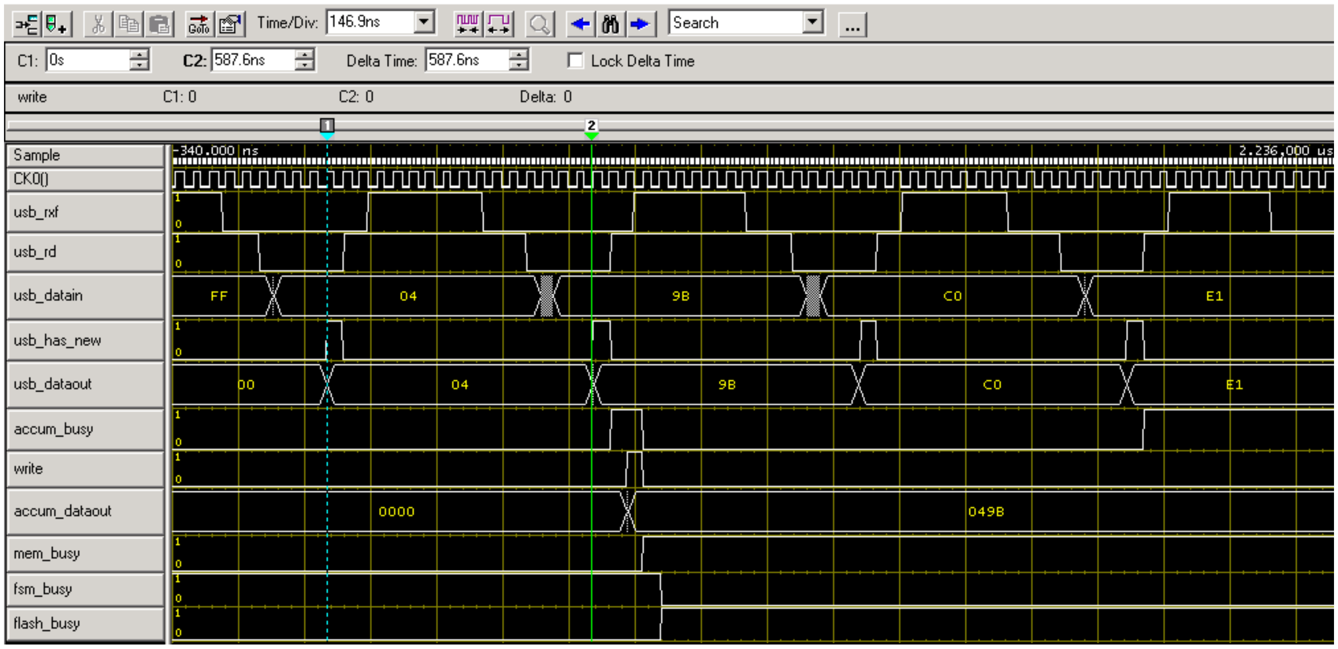


Figure 8. Logic Analyzer readout at the beginning of a write cycle

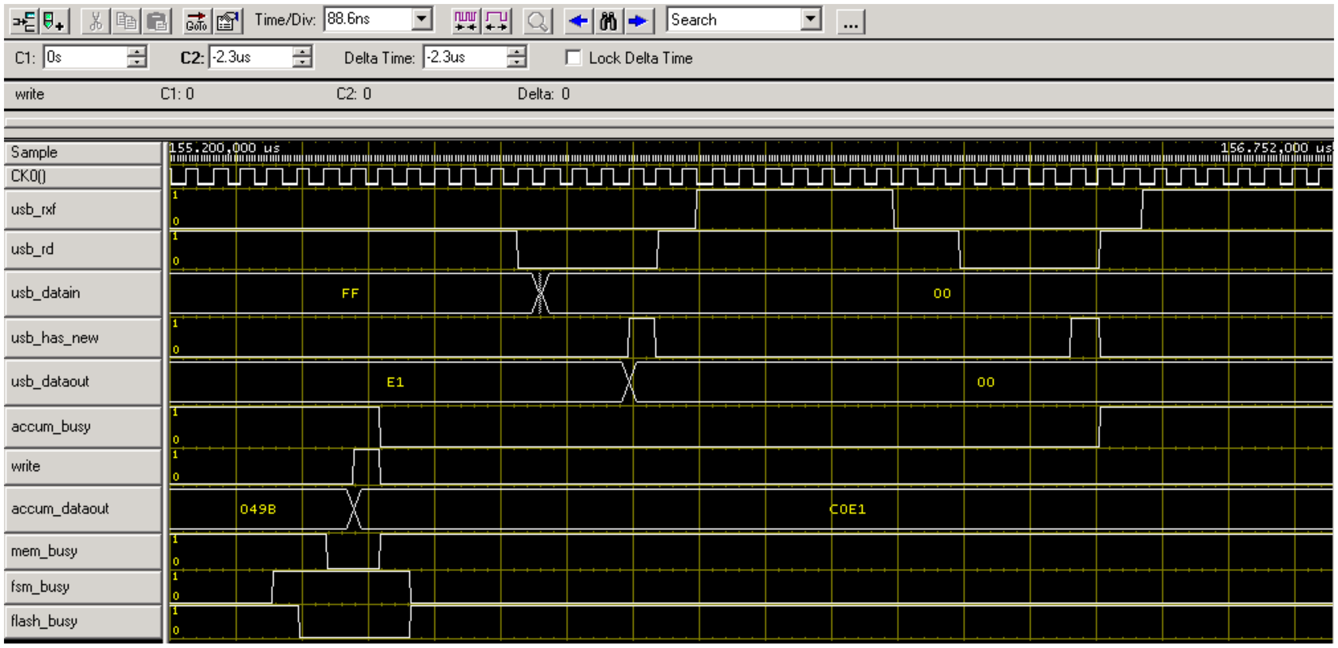


Figure 9. Logic analyzer readout at the end of a write cycle

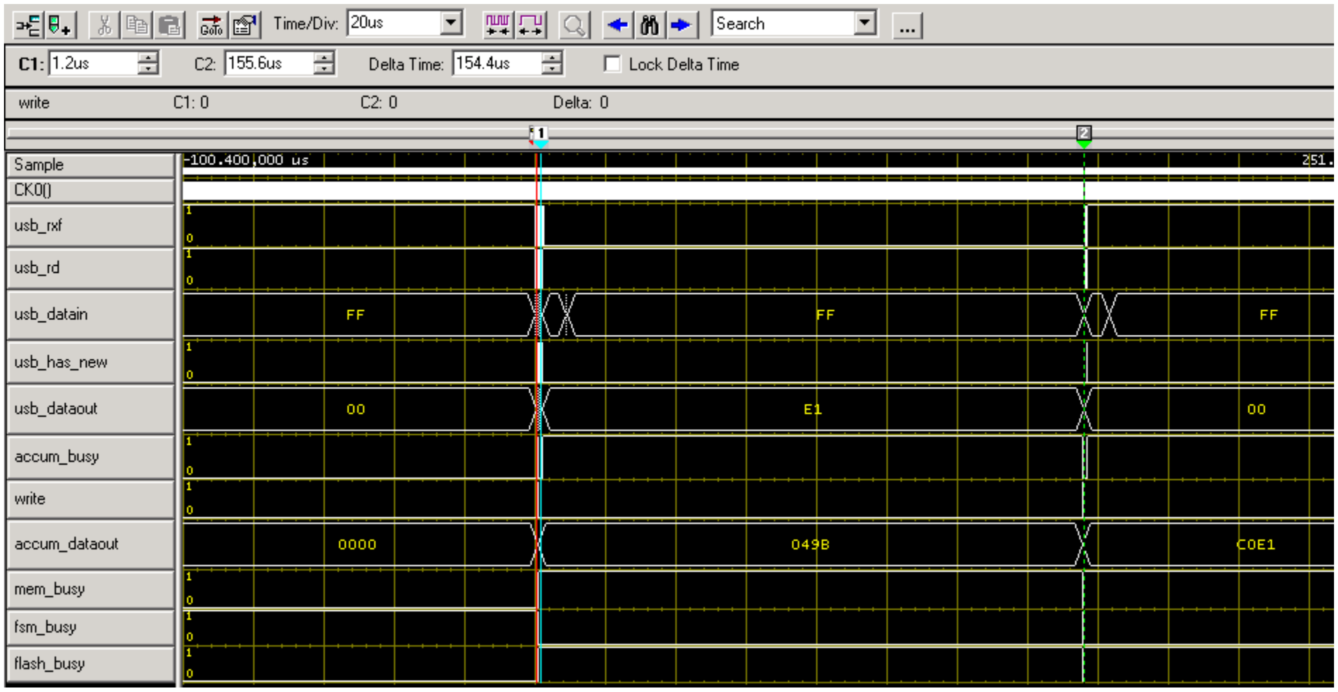
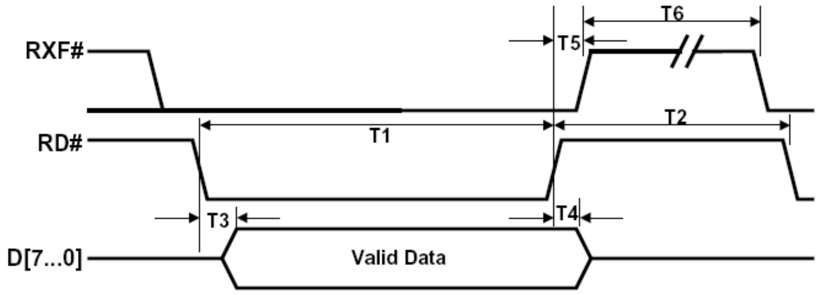


Figure 10. Logic analyzer readout showing time of total write

Score Data Transfer, Storage, and Playback: Write Mode Operation: usb_input
This module is responsible for communicating with the FTDI UM245 USB package. The FTDI UM245 maintains a simple FIFO queue, which can be queried as shown below in figure XXX.



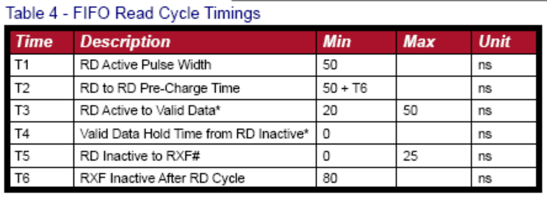


Table 4 - FIFO Read Cycle Timings

Time	Description	Min	Max	Unit
T1	RD Active Pulse Width	50		ns
T2	RD to RD Pre-Charge Time	50 + T6		ns
T3	RD Active to Valid Data*	20	50	ns
T4	Valid Data Hold Time from RD Inactive*	0		ns
T5	RD Inactive to RXF#	0	25	ns
T6	RXF Inactive After RD Cycle	80		ns

Figure 11. FTDI UM245 read specifications

Once the usb_input module has read a byte from the FIFO's queue, it lets outside observers know by raising its usb_has_new output to high and putting the byte read onto its output pins. The usb_input module only continues to read if its hold input is low, assuring that data is not read faster than it can be written.

Score Data Transfer, Storage, and Playback: Write Mode Operation: data_accum
Each address in the Flash ROM is two bytes wide, but the USB transfers data only one byte at a time.

The data_accumulator bridges this gap by concatenating bytes into 16 bit words, and then asking the flash_manager to write them.

Score Data Transfer, Storage, and Playback: Read Mode Operation
While in read mode, the driving signal is the beat signal from the user interface. On every transition of the beat signal (both rising and falling), the tempo_synth module updates its estimate of how the music should be played (actually a very difficult and complicated decision to make). The tempo synth module generates a square wave "tick" which alternates between high and low, with each transition representing an increment of time by one tick. As time increments, score_man executes notes from memory appropriately by enabling the correct instruments at the correct pitches at the correct time.

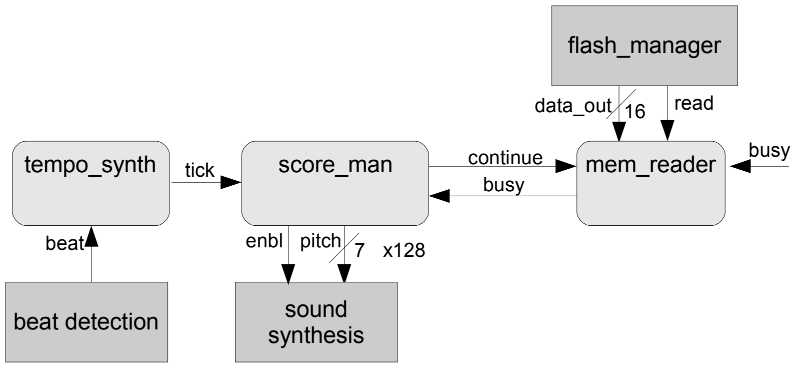
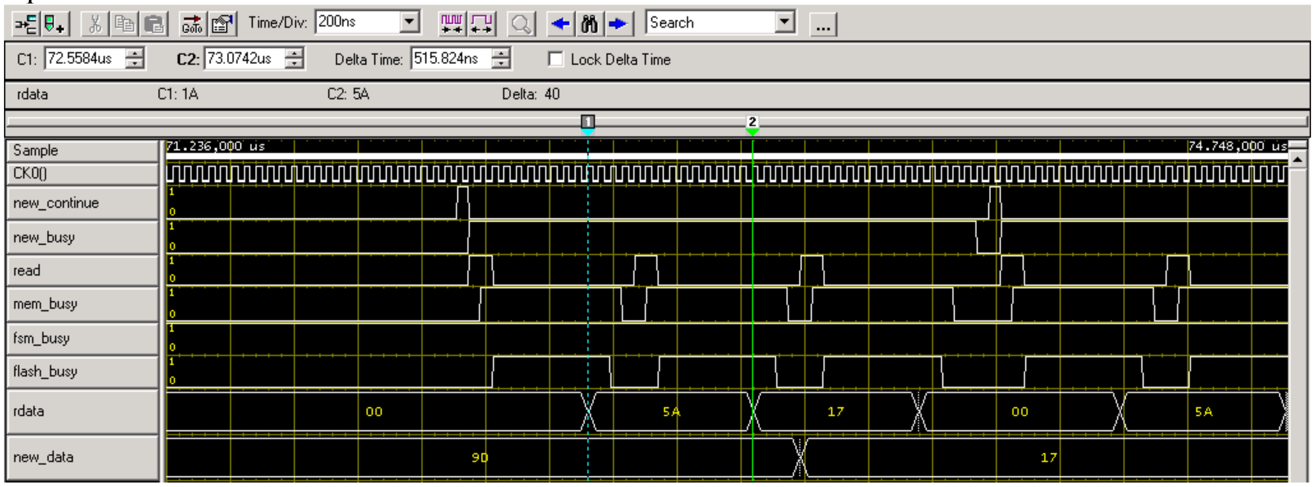


Figure 12. Block diagram of modules involved in reading

Performance wise, each read from memory takes 516ns, and three reads are required per note. When delays due to busy signals propagating back and forth are taken into account, executing a NoteData event takes 1.66us. This puts the note throughput at 602 kHz, which far exceeds minimum requirements.


Time/Div: 200ns

Search

C1: 72.5584us C2: 73.0742us Delta Time: 515.824ns Lock Delta Time

rdata C1: 1A C2: 5A Delta: 40

71.236,000 us 74.748,000 us

Figure 13. Zoomed in view of read cycle
(rdata is 2 bytes wide - only showing low order byte, new_data is 6 bytes wide – only showing 3rd byte)

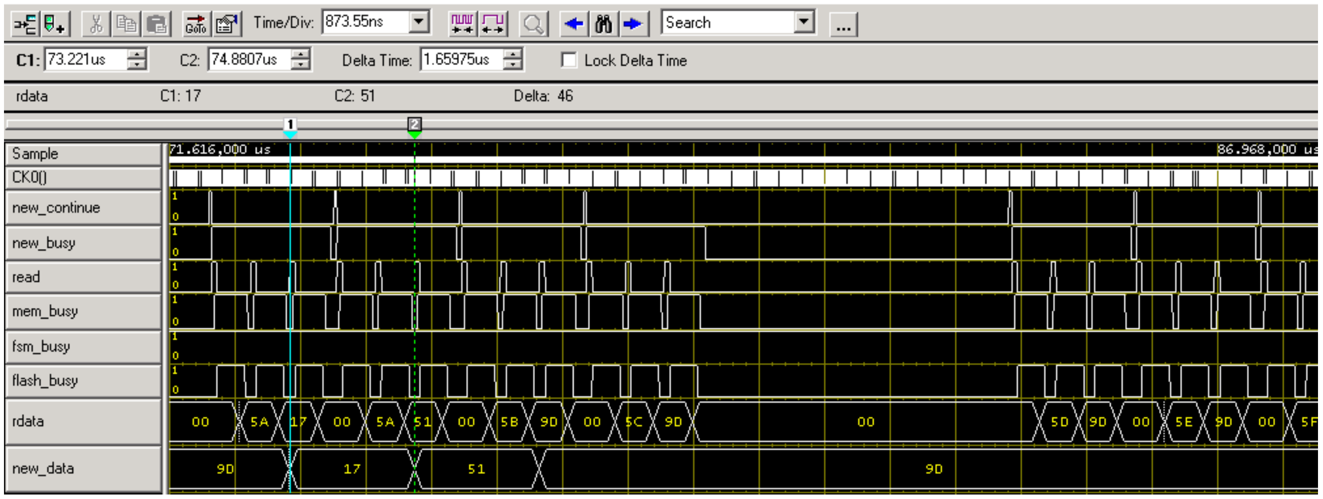


Figure 14. Zoomed out view of read cycle
(rdata is 2 bytes wide - only showing low order byte, new_data is 6 bytes wide – only showing 3rd byte)

Score Data Transfer, Storage, and Playback: Read Mode Operation: tempo_synth
We vastly underestimated the importance and difficulty of this module. In out original plan, this module would predict a tempo based on the delta-time from the beat, and stop counting forward once the beat was about to increment. In this way, the beat would always be perfectly synchronized to the conductor's motion, although tempo would be a little jerky. In practice, tempo was unacceptably jerky. When slowing down the score would play to the end of the beat much too fast and then wait for the next beat. When speeding up the score would skip the last notes in each beat.

At the last moment, Yuta Kuboyama saved the team by writing a module that is always perfectly in sync with the conductor's tempo, but it ignores his beat. Similar to the earlier module, it looks at the difference in time between beats to calculate tempo, but it makes no attempt to prevent the beat from incrementing without the conductors explicit approval. This is the module used in our demonstration. This has the advantage that the music plays smoothly and the user has the power to both accelerando and decelerando smoothly, but it has the disadvantage that the conductor's beat is usually out of phase with the music.
Writing a module that keeps both tempo and beat phase within acceptable limits was much harder than we expected, and is one of the system's most glaring weaknesses.

Score Data Transfer, Storage, and Playback: Read Mode Operation: score_man
This is one of the simplest modules in the reading system. It stores in memory a single NoteData event. Whenever time increments, score_man checks to see if the timestamp of the note in its memory is less than or equal to the actual current time. If it is, the module sets the appropriate enable and pitch output registers to the appropriate values. It then asks mem_reader for the next note in memory, which mem_reader caches. If several notes are supposed to happen simultaneously, they will actually happen sequentially in this system. However, the time between notes when the mem_reader cache is empty (worst case scenario 1.66us) is much less than the time between samples (20.83us). This means that the resultant sound will be exactly accurate so long as there are less than 12 simultaneously occuring note changes, and even if there are more than 12 changes then the 13th-24th changes will happen one sample late, and the 25th-36th changes will happen two samples late, etc. Basically, even if the system's ample 602 kHz note throughput is exceeded temporarily, the results will be imperceptible.

Score Data Transfer, Storage, and Playback: Read Mode Operation: mem_reader
Each NoteData event is 48 bits wide, but each memory address is only 16 bits wide. Rather than requiring the score_man module to keep track of memory addresses and execute each of the three read operations required per NoteData, mem_reader presents a simple handshake mechanism for score_man to interface with the score in memory. Furthermore, mem_reader caches NoteData events one event ahead, reducing the significance of memory read delays.

Audio Synthesis
The task of the Sound Synthesis block is to take in the instrument enables and pitches from the Score Manager module, the volume value from the Conductor module, and output an audio signal that is characterized by those parameters. All audio signals used in this block are signed. An overview of the Sound Synthesis block is shown below.

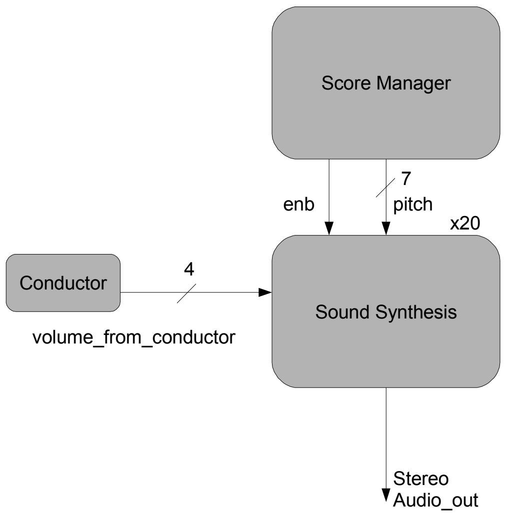


Figure 15. Overview of Sound Synthesis Module

Audio Synthesis: Orchestra
This is the top-level module of the Sound Synthesis block that is instantiated in the labkit. It encapsulates the entire Sound Synthesis block, to allow a smooth integration with the rest of the project. This makes the Sound Synthesis block very modular, and any changes made within the Sound Synthesis block would be invisible to the labkit or any other parts of the project. It consists of the instances of the DAC module, the Pulse Generator module, the four Instrument Manager modules (String, Brass, Woodwind, Percussion), and the Mixer module. The Orchestra module acts as an interface to the labkit, and therefore it will receive the enables and pitches from the Score Manager module, the volume value from the Conductor module, and applies those signals to the relevant internal modules. The Orchestra module also takes the output from the DAC module, and feeds it back into the labkit to be played out on the speakers. An overview of the orchestra module is shown below.

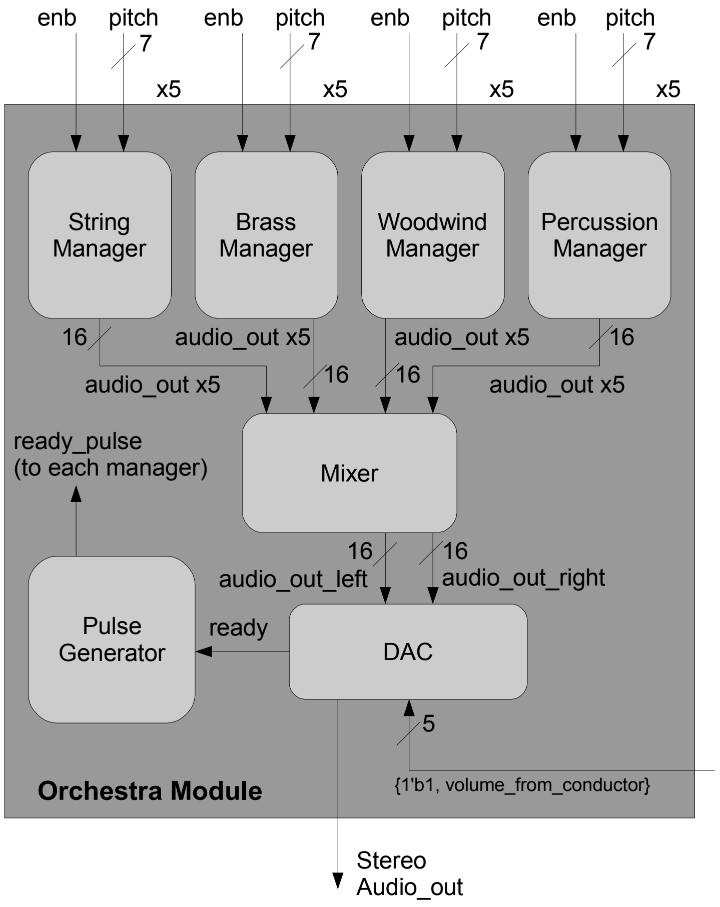


Figure 16. Orchestra Module

As the DAC module has a sample rate of 48kHz, when the 27MHz clock is used, there are 563 clock cycles between each ready pulse from the DAC that requests a new sample. Therefore, the diagram below shows the flow of processes in the Orchestra module between the ready pulses.

Audio Synthesis: Digital-to-Analog Converter (DAC)
This is a wrapper for the AC97 module. The AC97 module contained in this DAC module is a modified version of the AC97 from lab 4, which now supports 16-bit stereo audio instead of 8-bit mono. At a sample rate of 48kHz, it takes in the stereo 16-bit samples from the Mixer module, and sends them out to the speakers. Every time this module requests for a sample, it sets the ready signal high for a few clock cycles. Also, this module takes the volume value given by the conductor module, appends a 1'b1 as the most significant bit, and assigns it as the volume of AC97.

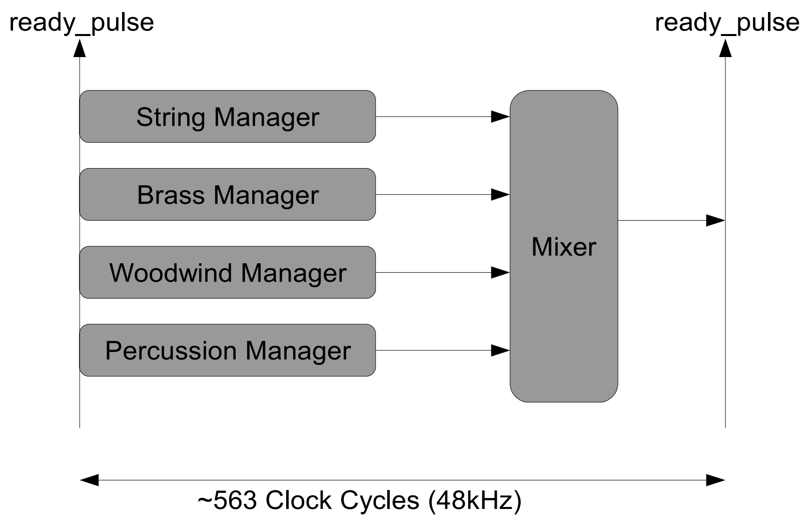


Figure 17. Orchestra Timing

Audio Synthesis: Pulse Generator

This is a simple step-to-pulse converter, which detects the rising edge of the ready signal from the DAC module, and outputs it as a ready_pulse with a width of one clock cycle. There is a variable last_ready, which keeps track of the ready signal in the previous clock cycle. Only when there is a transition from 0 to 1 in last_ready to ready, the ready_pulse is generated. This ready_pulse output is fed to all of the other Instrument Manager modules.

Audio Synthesis: Strings Manager/ Brass Manager

These two modules have identical architectures, except that the Strings Manager module uses string audio samples, and the Brass Manager module uses brass audio samples. These modules take in the enables and pitches of their instruments, and output their audio signals individually at the given pitch, while enable is high. They both contain five instruments, and each instrument consists of an Oscillator module and an Envelope module. The layout of each instrument is as follows.

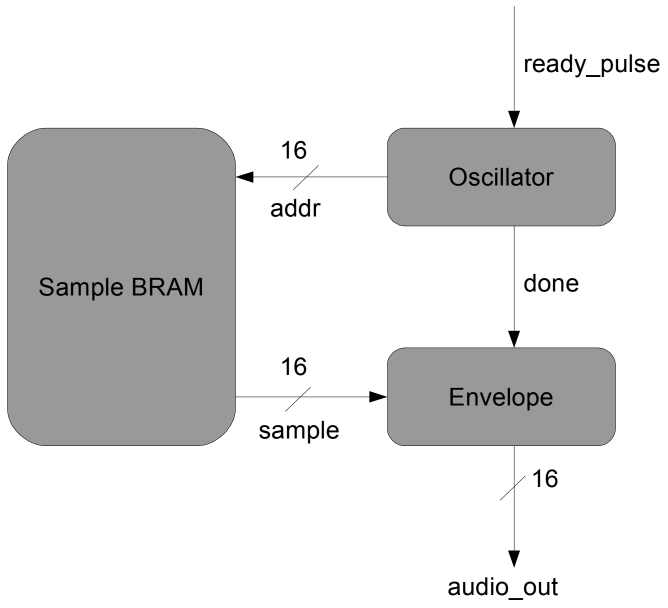


Figure 18. Instrument Structure

The Oscillator module calculates the address of the sample BRAM according to the pitch value it receives. This address is sent to the sample BRAM, and the obtained sample is fed into the Envelope module. If the oscillator_done signal is asserted, the Envelope module takes the sample from the BRAM, applies an amplitude envelope to the sample, and outputs it as audio_out if the enable signal is high.

These instruments are processed sequentially, using only one instance of the sample BRAM.

For example, in the case of the Strings Manager module, violin1 determines its signal, then violin 2, then viola, then cello, and finally the double bass.

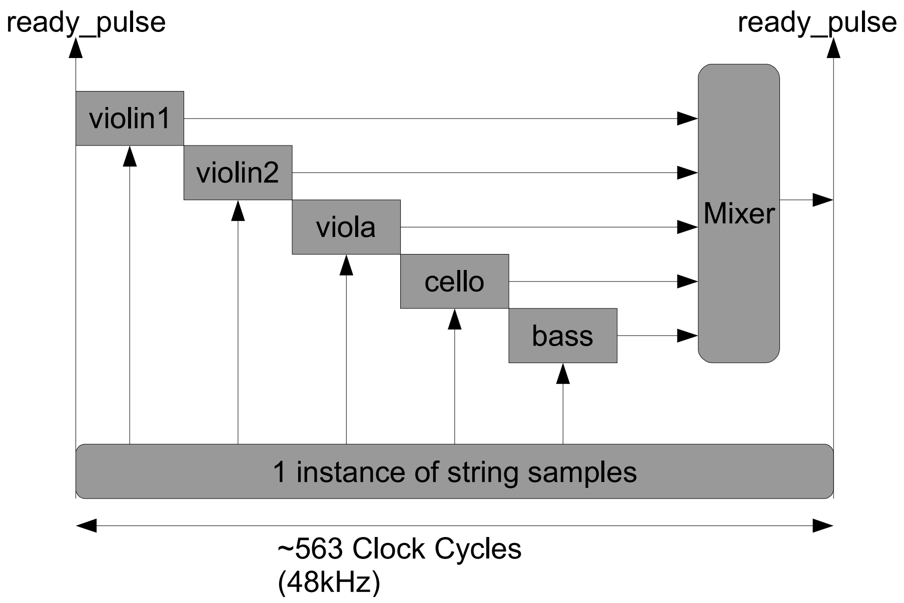


Figure 19. String Manager Timing

The reason for this type of operation is due to the constraints on the resources available on the FPGA. The FPGA contains 2.4Mbits of BRAM, which is not enough to store samples of each instrument individually. Therefore, there is a need to share a certain set of samples between different instruments. Fortunately, all string instruments (violin, viola, cello, double bass) have very similar waveforms, and all the brass instruments (trumpet, horn, trombone, tuba) have very similar waveforms as well. Therefore, it is possible to use the same set of samples for the strings and brass instruments. In addition, as mentioned before, the DAC sample rate (48kHz) is slower than the internal clock frequency (27MHz), and there are 563 clock cycles that can be used in between each sample request from the DAC. This provides enough time to produce the audio signals for each instrument sequentially using only one instance of the audio sample BRAM, instead of producing audio signals simultaneously using multiple instances of the audio sample BRAMs.

Audio Synthesis: Woodwinds Manager

This module is identical to the Strings/Brass Managers, except that instead of sharing a single sample BRAM, this module instantiates sample BRAMs for every instrument. This is because the audio waveforms of woodwind instruments (flute, clarinet, oboe, sax, basoon) are not very similar, and the same samples cannot be used to represent all of the instruments. Therefore, the processes in this module are simultaneous.

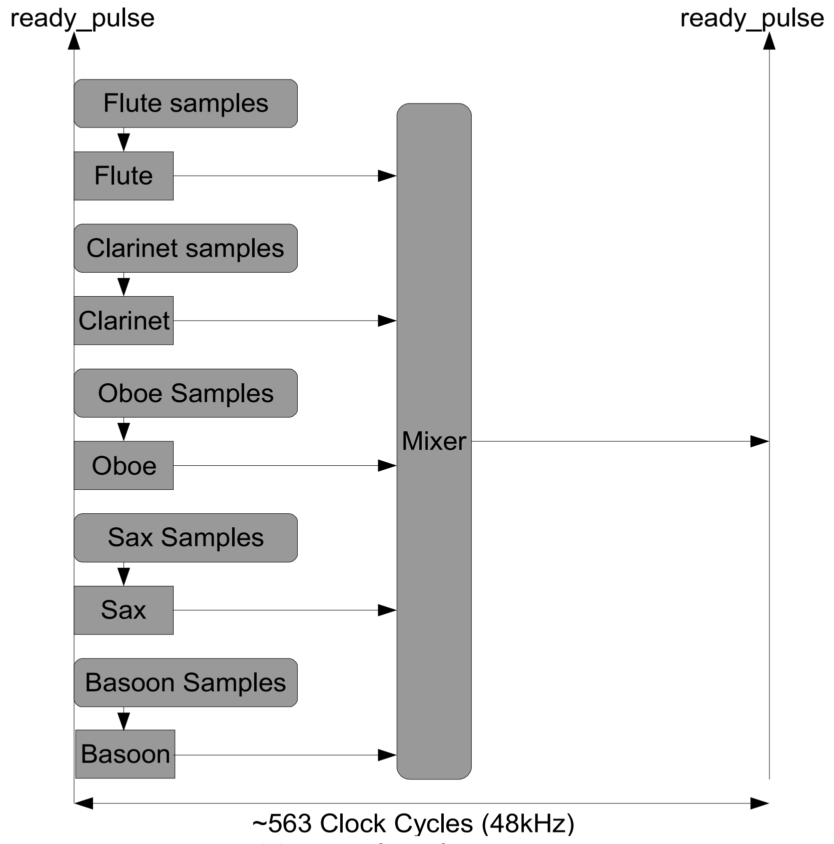


Figure 20. Woodwind Manager Timing

This module takes in the enables and pitches for flute, clarinet, oboe, sax, and basoon, and outputs their audio signals at the given pitch while enable is high.

Audio Synthesis: Percussion Manager

In this module, each instrument has its own sample BRAM, just like in the Woodwinds Manager module. However, this module differs from the other Instrument Managers, as percussion instruments do not have pitches (except timpani), and they are not periodic. All instruments except timpani must always be sampled at the same frequency, and the instruments should detect the rising edge of the enable and play until they go through their sample BRAMs. Percussion instruments are not on/off instruments, but their amplitude decay over time after the strike, so it must not turn on/off with the enable high/low, but instead detect the rising edge of the enable and decay over time.

For the constant sampling rate mechanism, a simple counter is used for every instrument, which counts up to a certain period value, and once it reaches that period value, the address value on the sample BRAM is incremented.

The percussion audio samples also contain their own ADSR envelopes, so it is not necessary to put them through the Envelope module. Therefore this module outputs the audio signal at the rising edges of the enable signal.

Audio Synthesis: Sample BRAM

The sample BRAMs are wave file samples that were converted into .coe files using MATLAB and initialized during the generation of the BRAM. Each address contains a signed 16-bit sample, and the number of samples depends on the sampled instrument. The samples are generated such that if the address of the sample BRAMs were incremented and looped through, the output audio would be a single continuous note.

Audio Synthesis: Oscillator
This module takes in a pitch, and looks at a period table BRAM to determine the period of the pitch (the number of clock cycles in between incrementing the sample table address), and determines the next address of the sample table to take the sample from. It does this by calculating how many times the period would fit in before the next ready pulse is asserted, and increments the sample address that number of times.

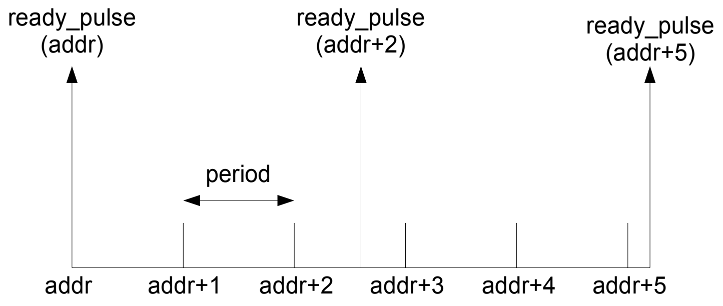


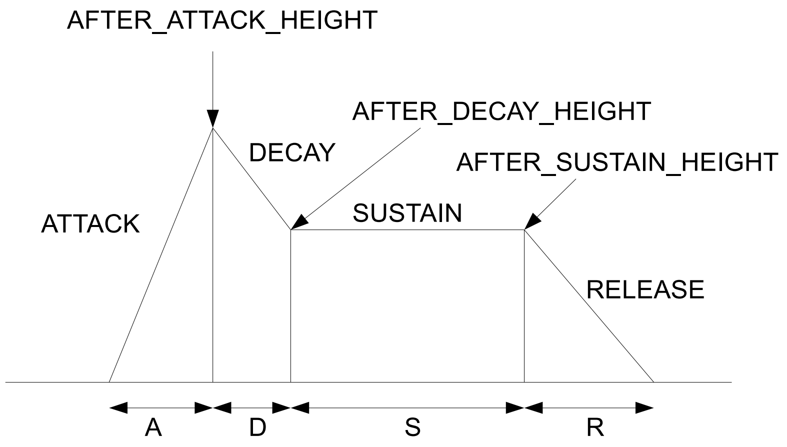
Figure 21. Oscillator Sample Lookup

This is the basic structure of the Oscillator module. In the actual implementation, different instruments have their own versions of the Oscillator (string_oscillator, oboe_oscillator, etc), as each instrument was sampled at different pitches, and the values in the period table BRAM differed from instrument to instrument.

Audio Synthesis: Period Table BRAM
The period table BRAM is a list of 20-bit period values such that when the pitch value is set as the address, the output of this memory is the period for that pitch. A period is the number of clock cycles it takes before incrementing the address on the sample BRAM. This means that the frequency at which the sample BRAM address will be incremented depends on the period, and altering the period will therefore alter the frequency of the audio signal. To generate this period table, MATLAB was used to create a .coe file that was loaded during the generation of the BRAM.

Audio Synthesis: Envelope
This module applies an ADSR (Attack, Decay, Sustain, Release) amplitude envelope to the audio sample. It takes in an oscillator_done signal, and when that signal is asserted, it first calculates the ADSR stage it is in. Then, the module receives a sample from the sample BRAM, and while the enable is asserted, it calculates the output audio signal with the appropriate amplitude envelope. The amplitude envelope is characterized by the ADSR parameters specified in the module. The DURATION parameters for ADSR determines how long each of the stage lasts for. The STEP parameters for ADSR determines how rapidly the amplitude increases/decreases per sample. The AMPLITUDE_RESOLUTION parameter determines the resolution of the envelope. The AFTER_ATTACK_HEIGHT, AFTER_DECAY_HEIGHT, and AFTER_SUSTAIN_HEIGHT

determines the amplitude of the signal after the respective ADSR stages. An ADSR envelope is shown below.



A – ATTACK_DURATION
D – DECAY_DURATION
S – SUSTAIN_DURATION
R – RELEASE_DURATION

Fig 22. ADSR Envelope

This module has a variable adsr_state, which can be either ATTACK, DECAY, SUSTAIN, RELEASE, or OFF. By looking at when the enable signal turns on, how long the enable signal is on for, and when it turns off, the adsr_state is updated. If the enable turns from low to high, the instrument will go through ATTACK, DECAY, and SUSTAIN. During any of these three states, if the enable goes low, the instrument will go into the RELEASE state, and once RELEASE_DURATION is reached, adsr_state will go back to OFF again. The sample obtained from the sample BRAM is multiplied by an integer and then bit-shifted to be able to gradually increase/decrease between 0 and 1.3 times the original amplitude of the sample. The adsr_timer variable keeps track of how long the instrument has been in that particular adsr_state.

The amplitudes in each adsr_state is calculated as follows:

ATTACK_STATE:
(((ATTACK_STEP*adsr_timer)*sample) >> AMPLITUDE_RESOLUTION);

DECAY_STATE:
(((AFTER_ATTACK_HEIGHT - (DECAY_STEP*adsr_timer))* sample)
>> AMPLITUDE_RESOLUTION);

SUSTAIN_STATE (if infinite sustain):
((AFTER_DECAY_HEIGHT * received_sample) >> AMPLITUDE_RESOLUTION);

SUSTAIN_STATE (finite sustain):
(((AFTER_DECAY_HEIGHT-(SUSTAIN_STEP*adsr_timer))*sample)
>> AMPLITUDE_RESOLUTION);

RELEASE_STATE:

```
(((AFTER_SUSTAIN_HEIGHT - (RELEASE_STEP*adsr_timer))* sample)
>> AMPLITUDE_RESOLUTION);
```

Audio Synthesis: Mixer
This module takes in all of the audio_out signals from the instrument managers, and adds them up into a single 24-bit stereo signal. The addition is done one at a time (ie one instrument per clock cycle), as adding all signals at once violates the timing constraints for the 27MHz clock. Once all of the samples are added together, the 24-bit value is scaled to 16-bits and sent out as a 16-bit stereo signal.

Testing and Debugging

Testing and Debugging: Video Tracking and Beat Detection
Creating the conductor module involved a lot of testing and debugging. Incremental testing proved to be very useful when building the conductor module – simulating the module to test and debug was unsuccessful because it was hard to pinpoint the exact problem in the modules. First, we had to make sure that the video data was buffered and outputted on the screen correctly. To test the change from YCrCb to RGB and HSV, we used the computer monitor to detect if we had the right red, green, and blue signals. HSV was harder to test; we ended up putting the hue to all the red, green, and blue signals, to see which objects had the greatest hue. While changing the video from YCrCb, RGB, and HSV, we used different shapes and colors to find what color and object would be best for our conductor module to detect. At first, we used LED lights, but soon found that the video camera could only detect it if it was not flashed directly at the video camera. This was quite cumbersome because the conductor would have to make sure to point it at another object. Furthermore, the LED light turned out to be too small for the video camera to detect clearly. Next, we tried an object with a flat circular surface. This detected well but was extremely hard to detect if the flat surface was pointed at an angle to the light. This was because the light changed the color of the object so much that changing the threshold range would allow other colors to seep into the outputs. Consequently, we chose a round circular object because there was always enough light that reflected onto the object that would be within the threshold. Figure 4 shows how effective it was to have a round pink surface to detect an object.

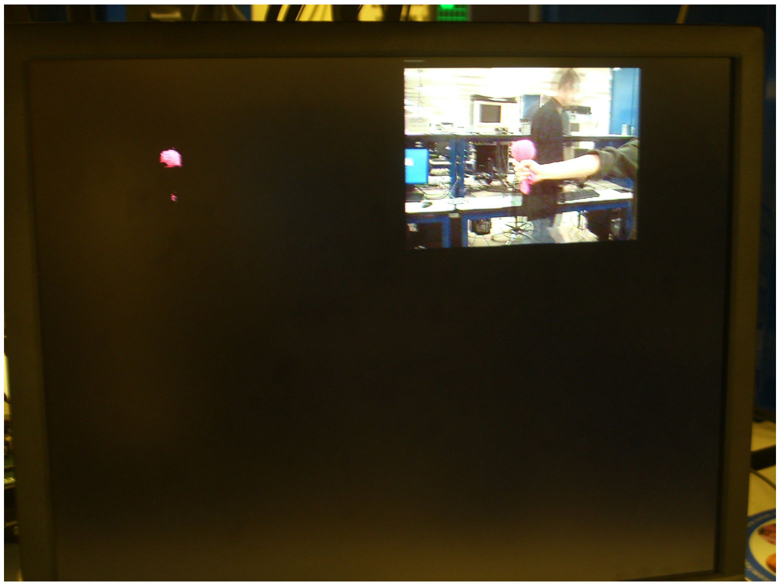


Figure 23. Threshold_finder output of the object (left) and the color video output (right).

A problem that we encountered was that the beat of the object would suddenly change immensely if the movements were not smooth. Based on the numerous tests while “conducting”, we realized that when we held the object in midair, our hand would naturally move around, thus causing the center of mass position of the object to slightly jump around. To fix this, we allowed a small range of movement for the center of mass coordinates so that our module would not detect the slightest of movements. To find the range, we used the 16 hex display to see what the range of x coordinates were when the object was stationary. The 16 hex display also proved useful when finding the volume of the piece. Because we had to test our modules individually, the hex display showed how the volume would change abruptly if the distance between beats was not smooth enough. Consequently, a volume controller module was created to slowly and smoothly increment and decrement the volume.

Another problem we faced with the beat counter was that it would detect many beats if we moved the baton one beat. Our problem turned out to be that our beat detector was looking at each clock cycle to find the x coordinate and compared it to the previous x coordinate of the center of mass. Because of this, the object would stabilize, but for the wrong reasons. We realized that the x coordinate needed to be taken every seven frames (about 1/10th of a second) so that the x coordinate would slightly change so that we could find a realistic difference in beats and volumes.

The LED, 16 hex display, and computer monitor turned out to be extremely useful to figure out any bugs in our modules. However, the logic analyzer was perfect in helping detect our errors in the beats and volume. It allowed us to see what was going on at every frame so that we could see if my beat had changed as well as if it was changing when switching directions. Using an LED light for this proved to be ineffective because the led light would flicker for a quick second and we would not have been able to figure out what had been going on.

Testing and Debugging: Score Data Transfer, Storage, and Playback
By far the biggest challenge was debugging the Flash memory. Even with 64 bits of data and write enables and FSM states on the logic analyzer, it was very difficult to determine what the correct behavior was. The hex display was very important in going through this step by step, finding errors.

The trickiest bug I ran into was that every 100 or so bytes of data transmitted over the USB, my usb_input state machine would randomly decide to read, even if there wasn't data being driven. As a result, I'd have one extra byte in in my bytestream which shifted everything to the left and ruined everything. I debugged and debugged and debugged and finally caught the glitch on the logic analyzer. As far as I could tell, my state machine was utterly violating the rules it was coded by, but only once every now and then. Thank God for Gim, who put capacitors on the power supply to the FTDI USB interface chip, and all of a sudden it was more like 1 in a 1000. This was good for small scores, but larger scores had to be sent several times before they arrived intact. This was unacceptable, but was finally fixed for good when I arbitrarily added several states to my state machine where it has to check and wait double check and wait triple check before it moves moves forward, and there hasn't been a single glitch since. I still have no idea why it was wigging out in the first place though.

Testing and Debugging: Audio Synthesis
Testing and verification took place after every new addition to the Sound Synthesis block, by playing back the audio samples through headphones/speakers. At first, the lab4 AC97 module was modified to support 16-bit stereo signals, and this was tested using the 750Hz tone module in lab4. Then, a violin

sample was imported into a BRAM, and was played back at a fixed pitch. Then, a push button was assigned as an enable signal to the violin, and was used to test the functionality of the enable signal on the violin. After the basic playback functionality was verified, the oscillator module was added, and the switches were assigned as pitch values to test whether the violin sound was played back at the specified pitches. Once this was done, the envelope module was added, and the ADSR duration was exaggerated to test the module. Once all of this was working for one instrument, the next step was to create the string manager with 5 string instruments. The string manager was verified by playing the 5 strings together at different pitches, and also playing each instrument separately by assigning more buttons as enables. This was then extended to the brass, woodwinds, and percussions, and they were all tested in a similar manner. Once all instrument managers were functional, the mixer module was written to combine all of the audio outputs together. More complex testing took place by integrating the orchestra module with the Score Manager module, and playing back certain MIDI files at constant tempo.

Testing and Debugging: System Integration
This was a very frustrating process. Each compile took upwards of 40 minutes. We had originally designed the system to be on three separate labkits, but in the end we each used different resources so there was no reason why we couldn't integrate onto one. Our interfaces were very well defined and very small, but when we compiled onto the same labkit everyone's modules went bonkers. The video stopped displaying, the USB stopped talking, and the audio system played random noises at random times. We tried and tried (and waited for compiles and waited for compiles) but had no luck.

When we compiled the exact same code with wires between two labkits, it worked the first time. However, it also exposed a weakness our tempo detection had to tiny and infrequent jitters in the beat detection. When two beats came right after each other, the tempo approaches infinity and the whole song plays in a few milliseconds.

Overall, we felt very good that our project worked the first time (with wires), but we wish we had realized the problems with our beat-to-tempo conversion earlier.

Conclusion
The objective of this project was to be able to conduct a synthesized orchestra using hand motion and gestures, and we believe that this functionality was implemented successfully. The conductor module has managed to detect beat and dynamics from user hand gestures. The score manager module is capable of managing the instruments accurately to play the stored scores. The orchestra module produces audio signals that resemble a real orchestra. Possible next steps for this project may be the addition of section controls of the orchestra, improvement of tempo controls, and the addition of audio effects such as reverb.

References

Terman, Chris. 6.111. Vers. Fall 2007. Aug. 2007. Massachusetts Institute of Technology. 10 Oct. 2007 <http://web.mit.edu/6.111/www/f2007/index.html>.

"Timing Tool – the Timing Diagram Editor." July 2007. Timing Tool. 10 Oct. 2007 <http://www.timingtool.com/menu>.

Ickes, Nathan. Flash Tester Modules. January 2005. Massachusetts Institute of Technology. <http://www-mtl.mit.edu/Courses/6.111/labkit/verilog/004/flashtest/doc/flashtest.shtml>.

Denckla, Ben and Pelletier, Patrick. Rogus McBogus. Massachusetts Institute of Technology. <http://www.media.mit.edu/hyperins/rogus/home.html>.

UM245 Data Sheet. FTDI. <http://www.ftdichip.com>.

Appendix A: Video Tracking and Beat Detection: beat_counter.v

```
module beat_counter(clk, reset, hcount, vcount, new_x, beat, count, east, ready, 
pulse,
                                          n_clock, x_init, prev_x, west, volume, 
volume_clean);
input clk, reset;
input [10:0] new_x, hcount;
input [9:0] vcount;
output beat;
output [7:0] count;
output ready, east, pulse, west;
output n_clock;
output [11:0] x_init, prev_x;
output [3:0] volume;
output [3:0] volume_clean;

wire [3:0] volume_clean;

reg beat, pulse, east, west;
reg [10:0] x, x2, x3, x4, x5, x6, x7, x_init, x_beat, x_beat_prev;
reg [7:0] count;
reg n_clock, change;
reg [3:0] volume;

assign ready = (count == 180);//after calibrating
assign prev_x = x7;

always @ (posedge clk) begin
if (reset) begin
beat <= 1'b0;
x <= new_x;
x2 <= 11'b0;
x3 <= 11'b0;
x4 <= 11'b0;
x5 <= 11'b0;
x6 <= 11'b0;
x7 <= 11'b0; // to save the past seven frames
x_init <= 11'b0;
count <= 8'b0;
pulse <= 1'b0; // just to make sure the beat changes once
east <= 1'b0;
west <= 1'b0;
change <= 1'b0; // just to make sure the direction changes once
volume <= 4'd8;
x_beat <= 11'b0;
x_beat_prev <= 11'b0;
end

if ((vcount == 767) && (hcount == 1023)) begin
n_clock <= ~n_clock; // debugging tool
            x <= new_x;
            x2 <= x;
            x3 <= x2;
            x4 <= x3;
            x5 <= x4;
            x6 <= x5;
            x7 <= x6;
```

```
      if ((count < 180) && (x >= x7) && ~(x == 2047)) begin   //stablize time
                  if (x <= 5 + x7) count <= count + 1;
                        else  count <= 8'b0;
                  end
      if ((count < 180) && (x < x7) && ~(x == 2047)) begin
                  if (x7 <= 5 + x)  count <= count + 1;
                        else  count <= 8'b0;
                  end
//////////////
if (count == 170) x_init <= new_x;

if (ready) begin
      if ((x > x_init) && (x >= x_init + 7) && ~change) begin east <= 1'b1;

                              change <= 1'b1; end// change so only one instance
      if ((x_init > x) && (x_init > x + 7) && ~change) begin west <= 1'b1;

                              change <= 1'b1; end
end

if (east | west) begin // already created a direction
      if (ready && east) begin
                  if (x < prev_x) begin // maybe going west or just a glitch
                        if ((x + 5 <= prev_x) && ~pulse) begin// going west for the
first time after going east
                              east <= 1'b0;
                              west <= 1'b1;
                              beat <= ~beat;
                              pulse <= 1'b0;
                              volume <= (x_beat_prev == 0) ? 4'd15 : ((x_beat >=
x_beat_prev) ? ((x_beat - x_beat_prev) >> 4) : ((x_beat_prev - x_beat) >> 4));
                              x_beat <= x;
                              x_beat_prev <= x_beat;
                              end
                        if ((x + 5 <= prev_x) && pulse) begin // going west , beat
has already changed
                              beat <= beat;
                              pulse <= 1'b0;
                        end
                  end
                  if (x >= prev_x) beat <= beat;
      end
      if (ready && west) begin
                  if (x > prev_x) begin// maybe going east or just a glitch
                        if ((prev_x + 5 <= x) && ~pulse) begin// going east for the
first time after going west
                              east <= 1'b1;
                              west <= 1'b0;
                              beat <= ~beat;
                              pulse <= 1'b1;
                              volume <= (x_beat_prev == 0) ? 4'd15 : ((x_beat >=
x_beat_prev) ? ((x_beat - x_beat_prev) >> 4) : ((x_beat_prev - x_beat) >> 4));
                              x_beat <= x;
                              x_beat_prev <= x_beat;
                              end
                        if ((prev_x + 5 <= x) && pulse) begin // going east , beat
```

```
has already changed
                              beat <= beat;
                              pulse <= 1'b1;
                        end
                  end
                        if (x <= prev_x) beat <= beat;
      end
end
end

end

//wire [3:0] volume_clean;
//
//    //module volume_control(clk, reset, noisy, clean);
volume_control volumecontrol1(n_clock, reset, volume, volume_clean);
endmodule
```

Appendix A: Video Tracking and Beat Detection: button_coordinates.v

```
module button_coordinates(clk, reset, hcount, vcount, up, down, left, right, x, y);
input clk, reset, up, down, left, right;
input [10:0] hcount;
input [9:0] vcount;
output [10:0] x;
output [9:0] y;


reg [10:0] x;
reg [9:0] y;

always @ (posedge clk) begin
if (reset) begin
x <= 11'd50;
y <= 10'd50;
end

if ((hcount == 1023) && (vcount == 765)) begin
y <= (up && (y >= 0)) ? y - 1 : y;
y <= (down && (y <= 765)) ? y + 1 : y;

x <= (right && (x <= 1020)) ? x + 1 : x;
x <= (left && (x >= 0)) ? x - 1 : x;
end
end
endmodule
```

Appendix A: Video Tracking and Beat Detection: conductor.v

```
module conductor(clock_27mhz, reset, tv_in_line_clock1, tv_in_ycrcb, switch0,
switch1, switch6, switch7, vga_out_red,
                              vga_out_blue,  vga_out_green, beats, tv_in_reset_b,
vram_read_data, ram0_data, ram0_clk, ram0_we_b,
                              ram0_address, ram0_cen_b, vga_out_sync_b,
vga_out_blank_b, vga_out_hsync, vga_out_vsync, vga_out_pixel_clock,
tv_in_i2c_clock, tv_in_i2c_data,
                              counter, hcount_avg, vcount_avg, clk, east, ready,
pulse, n_clock, x_init, prev_x, west, volume, volume_clean, beat_counter);
```

```
input  reset, clock_27mhz, tv_in_line_clock1,
                        switch0, switch1, switch6, switch7;
input  [19:0] tv_in_ycrcb;
output [7:0] vga_out_red, vga_out_blue, vga_out_green;
output beats, tv_in_reset_b;
output [7:0] counter;
output [35:0] vram_read_data, ram0_data; // data read from memory
output      ram0_clk, ram0_we_b;    // physical line to ram we_b
output [18:0] ram0_address;   // physical line to ram address
output       ram0_cen_b;      // physical line to ram clock enable
output vga_out_sync_b, vga_out_blank_b, vga_out_hsync, vga_out_vsync,
vga_out_pixel_clock;
output tv_in_i2c_clock, tv_in_i2c_data;
output [10:0] hcount_avg,  x_init, prev_x;
output [9:0] vcount_avg;
output clk;
output [3:0] volume_clean, volume;
output east, ready, pulse, n_clock, west;
output [3:0] beat_counter;

   ////////////////////////////////////////////////////////////////////////////
   // Demonstration of ZBT RAM as video memory

   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

   wire clk = clock_65mhz;

   wire right, left, down, up;

      debounce db2(power_on_reset, clk, ~button_right, right);
      debounce db3(power_on_reset, clk, ~button_left, left);
      debounce db4(power_on_reset, clk, ~button_down, down);
      debounce db5(power_on_reset, clk, ~button_up, up);

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


   // wire up to ZBT ram

   wire [35:0] vram_write_data;
   wire [35:0] vram_read_data;
   wire [18:0] vram_addr;
   wire        vram_we;
//module zbt_6111(clk, cen, we, addr, write_data, read_data,
//            ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);
//
//   input clk;                // system clock
//   input cen;                // clock enable for gating ZBT cycles
//   input we;                 // write enable (active HIGH)
//   input [18:0] addr;        // memory address
//   input [35:0] write_data;  // data to write
//   output [35:0] read_data;  // data read from memory
//   output        ram_clk;    // physical line to ram clock
//   output        ram_we_b;   // physical line to ram we_b
//   output [18:0] ram_address;     // physical line to ram address
//   inout [35:0]  ram_data;   // physical line to ram data
//   output        ram_cen_b;  // physical line to ram clock enable
   zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
            vram_write_data, vram_read_data,
            ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

   // generate pixel value from reading ZBT memory
   wire [17:0]    vr_pixel;
   wire [18:0]    vram_addr1;

   vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
             vram_addr1,vram_read_data);

   // ADV7185 NTSC decoder interface code
   // adv7185 initialization module

   adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
                .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
                .tv_in_i2c_clock(tv_in_i2c_clock),
                .tv_in_i2c_data(tv_in_i2c_data));

   wire [29:0] ycrcb;   // video data (luminance, chrominance)
   wire [2:0] fvh;      // sync for field, vertical, horizontal
   wire       dv; // data valid

   ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                .ycrcb(ycrcb), .f(fvh[2]),
                .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

      // code to write NTSC data to video memory

   wire [18:0] ntsc_addr;
   wire [35:0] ntsc_data;
   wire        ntsc_we;
      reg [9:0] O;
      wire [7:0] R,G,B,new_red, new_green, new_blue;


      YCrCb2RGB ycrcb_to_rgb( R, G, B, clk, reset, ycrcb[29:20], ycrcb[19:10],
ycrcb[9:0]);


      ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, {R[7:2], G[7:2], B[7:2]},
                ntsc_addr, ntsc_data, ntsc_we, switch6);
```

```
//    color_finder finder(clk, reset, {vr_pixel[17:12], 2'd0}, {vr_pixel[11:6],
2'd0}, {vr_pixel[5:0], 2'd0},
//                                                      new_red, new_green,
new_blue);

   // code to write pattern to ZBT memory
   reg [31:0]     count;
   always @(posedge clk) count <= reset ? 0 : count + 1;

   wire [18:0]    vram_addr2 = count[0+18:0];
   wire [35:0]    vpat = ( switch1 ? {8{4'b0}}
                   : 32'b0 );

   // mux selecting read/write to memory based on which write-enable is chosen

   wire     sw_ntsc = ~switch7;
   wire     my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank;
   wire [18:0]    write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
   wire [35:0]    write_data = sw_ntsc ? ntsc_data : vpat;

//   wire   write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;
//   assign       vram_addr = write_enable ? write_addr : vram_addr1;
//   assign       vram_we = write_enable;

   assign   vram_addr = my_we ? write_addr : vram_addr1;
   assign   vram_we = my_we;
   assign   vram_write_data = write_data;

            // select output pixel data
      wire [7:0] H, S, V;
      wire [13:0] ans;
      wire done;

      rgb_to_hsv hsv({vr_pixel[17:12], 2'd0}, {vr_pixel[11:6], 2'd0},
{vr_pixel[5:0], 2'd0}, clk, reset, H, S, V, ans, done);

   reg [17:0]     pixel;
   wire     b,hs,vs;

   delayN dn1(clk,hsync,hs);  // delay by 3 cycles to sync with ZBT read
   delayN dn2(clk,vsync,vs);
   delayN dn3(clk,blank,b);

   always @(posedge clk)
     begin
      pixel <= switch0 ? {hcount[8:0],9'b0} : vr_pixel;
     end

      wire [10:0] x_button;
      wire [9:0] y_button;

      button_coordinates buttons(clk, reset, up, down, left, right, x_button,
y_button);


      wire [10:0] x_crhair;
      wire [9:0] y_crhair;
      wire [10:0] hcount_avg, x, prev_x,  x_init;
      wire [9:0] vcount_avg;
      wire [7:0] red_coord, green_coord, blue_coord;

      RGB_value_finder rgb(clk, reset, hcount, vcount, x_crhair, y_crhair,
{vr_pixel[17:12], 2'd0}, {vr_pixel[11:6], 2'd0}, {vr_pixel[5:0], 2'd0},
                                                red_coord, green_coord,
blue_coord);



      threshold_finder thresholdfinder(clk, reset, hcount, vcount,
                                                     {vr_pixel[17:12], 2'd0},
{vr_pixel[11:6], 2'd0}, {vr_pixel[5:0], 2'd0}, new_red,
                                                     new_green, new_blue,
hcount_avg, vcount_avg);

      wire beats;
      wire [7:0] counter;
      wire east, west, ready, pulse, n_clock;
      wire [3:0] volume_clean, volume;
      //module beat_counter(clk, reset, hcount, vcount, new_x, beat, count, east,
ready, pulse, n_clock);
      beat_counter beat(clk, reset, hcount, vcount, hcount_avg, beats, counter,
east, ready, pulse, n_clock, x_init, prev_x, west, volume, volume_clean);


      reg beat_prev;
      reg [3:0] beat_counter;

      always @ (posedge clk) begin
      beat_counter <= (beat_prev == beats) ? beat_counter : beat_counter + 1;
      beat_prev <= beats;
      end


   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz. vga_out is 8 bits
      assign vga_out_red = switch1 ? (((hcount == (1024 - 3*hcount_avg)) || (vcount
== 3*vcount_avg)) ? 8'd255: 8'b0  ) : ((hcount >= 512) ?  {vr_pixel[17:12], 2'd0}:
8'b0);
      assign vga_out_green =switch1 ? (((hcount == (1024 - 3*hcount_avg)) ||
(vcount == 3*vcount_avg)) ? 8'd255 : 8'b0  ) : ((hcount >= 512) ? {vr_pixel[11:6],
2'd0}: 8'b0);
      assign vga_out_blue =switch1 ? (((hcount == (1024 - 3*hcount_avg)) || (vcount
== 3*vcount_avg)) ? 8'd255 : 8'b0 ) : ((hcount >= 512) ? {vr_pixel[5:0], 2'd0}:
8'b0);

   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_blank_b = ~b;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;


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
```

Appendix A: Video Tracking and Beat Detection: debounce.v

```
///////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module
//
///////////////////////////////////////////////////////////////////////////////
module debounce (reset, clk, noisy, clean);
   input reset, clk, noisy;
   output clean;
   parameter NDELAY = 650000;
   parameter NBITS = 20;

   reg [NBITS-1:0] count;
   reg xnew, clean;

   always @(posedge clk)
     if (reset) begin xnew <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != xnew) begin xnew <= noisy; count <= 0; end
     else if (count == NDELAY) clean <= xnew;
     else count <= count+1;
endmodule
```

Appendix A: Video Tracking and Beat Detection: display_16hex.v

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

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

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

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

   reg [5:0] count;
   reg [7:0] reset_count;
//   reg         old_clock;
   wire      dreset;
   wire      clock = (count<27) ? 0 : 1;

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

   assign dreset = (reset_count != 0);
   assign disp_clock = ~clock;
   wire   clock_tick = ((count==27) ? 1 : 0);
//   wire   clock_tick = clock & ~old_clock;

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

   reg [7:0] state;           // FSM state
   reg [9:0] dot_index;       // index to current dot being clocked out
   reg [31:0] control;        // control register
   reg [3:0] char_index;      // index of current character
   reg [39:0] dots;           // dots for a single digit
   reg [3:0] nibble;          // hex nibble of current character
   reg [63:0] data;

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

   always @(posedge clock_27mhz)
     if (clock_tick)
       begin
        if (dreset)
          begin
             state <= 0;
             dot_index <= 0;
             control <= 32'h7F7F7F7F;
          end
        else
          casex (state)
            8'h00:
            begin
               // Reset displays
               disp_data_out <= 1'b0;
               disp_rs <= 1'b0; // dot register
               disp_ce_b <= 1'b1;
               disp_reset_b <= 1'b0;
               dot_index <= 0;
               state <= state+1;
            end

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

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

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

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

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

            8'h06:
            begin
               // Load the user's dot data into the dot reg, char by char
               disp_rs <= 1'b0;                 // Select the dot register
```

```
              disp_ce_b <= 1'b0;
              disp_data_out <= dots[dot_index]; // dot data from msb
              if (dot_index == 0)
                  if (char_index == 0)
                    state <= 5;                // all done, latch data
                else
                  begin
                   char_index <= char_index - 1;     // goto next char
                   data <= data_in;
                   dot_index <= 39;
                  end
              else
                dot_index <= dot_index-1;      // else loop thru all dots
           end

         endcase // casex(state)
      end

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

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

```
endmodule
```

Appendix A: Video Tracking and Beat Detection: divider.v

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                 *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2006 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file divider.v when simulating
// the core, divider. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module divider(
      clk,
      dividend,
      divisor,
      quotient,
      remainder,
      rfd);


input clk;
input [22 : 0] dividend;
input [12 : 0] divisor;
output [22 : 0] quotient;
output [12 : 0] remainder;
output rfd;

// synopsys translate_off

      DIV_GEN_V1_0 #(
            1,    // algorithm_type
            0,    // bias
            0,    // c_has_aclr
            0,    // c_has_ce
            0,    // c_has_sclr
            0,    // c_sync_enable
            1,    // divclk_sel
            23,   // dividend_width
            13,   // divisor_width
            8,    // exponent_width
            0,    // fractional_b
            13,   // fractional_width
            1,    // latency
            8,    // mantissa_width
            0)    // signed_b
      inst (
            .CLK(clk),
            .DIVIDEND(dividend),
            .DIVISOR(divisor),
            .QUOTIENT(quotient),
            .REMAINDER(remainder),
            .RFD(rfd),
            .CE(),
            .ACLR(),
            .SCLR(),
            .DIVIDEND_MANTISSA(),
            .DIVIDEND_SIGN(),
            .DIVIDEND_EXPONENT(),
            .DIVISOR_MANTISSA(),
            .DIVISOR_SIGN(),
            .DIVISOR_EXPONENT(),
            .QUOTIENT_MANTISSA(),
            .QUOTIENT_SIGN(),
            .QUOTIENT_EXPONENT(),
            .OVERFLOW(),
            .UNDERFLOW());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of divider is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of divider is "black_box"

endmodule
```

Appendix A: Video Tracking and Beat Detection: ntsc2zbt.v

```
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.

///////////////////////////////////////////////////////////////////////////////
// Prepare data and address values to fill ZBT memory with NTSC data

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw);

   input     clk; // system clock
   input     vclk;      // video clock from camera
   input [2:0]     fvh;
   input     dv;
   input [17:0]    din;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output    ntsc_we;   // write enable for NTSC data
   input     sw;        // switch which determines mode (for debugging)

   parameter       COL_START = 10'd0;
   parameter       ROW_START = 10'd0;

   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 x 768 XGA display

   reg [9:0]       col = 0;
   reg [9:0]       row = 0;
   reg [17:0]      vdata = 0;
   reg             vwe;
   reg             old_dv;
   reg             old_frame; // frames are even / odd interlaced
   reg             even_odd;  // decode interlaced frame to this wire

   wire      frame = fvh[2];
   wire      frame_edge = frame & ~old_frame;

   always @ (posedge vclk) //LLC1 is reference
     begin
      old_dv <= dv;
      vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
      old_frame <= frame;
      even_odd = frame_edge ? ~even_odd : even_odd;

      if (!fvh[2])
        begin
           col <= fvh[0] ? COL_START :
                (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
           row <= fvh[1] ? ROW_START :
                (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
           vdata <= (dv && !fvh[2]) ? din : vdata;
```

```
         end
      end

    // synchronize with system clock

    reg [9:0] x[1:0],y[1:0];
    reg [17:0] data[1:0];
    reg       we[1:0];
    reg           eo[1:0];

    always @(posedge clk)
      begin
       {x[1],x[0]} <= {x[0],col};
       {y[1],y[0]} <= {y[0],row};
       {data[1],data[0]} <= {data[0],vdata};
       {we[1],we[0]} <= {we[0],vwe};
       {eo[1],eo[0]} <= {eo[0],even_odd};
      end

    // edge detection on write enable signal

    reg old_we;
    wire we_edge = we[1] & ~old_we;
    always @(posedge clk) old_we <= we[1];

    // shift each set of four bytes into a large register for the ZBT

    reg [35:0] mydata;
    always @(posedge clk)
      if (we_edge)
        mydata <= { mydata[17:0], data[1] };

    // compute address to store data in

    wire [18:0] myaddr = {1'b0, y[1][8:0], eo[1], x[1][9:2]};

    // alternate (256x192) image data and address
    wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};
    wire [18:0] myaddr2 = {1'b0, y[1][8:0], eo[1], x[1][7:0]};

    // update the output address and data only when four bytes ready

    reg [18:0] ntsc_addr;
    reg [35:0] ntsc_data;
    wire       ntsc_we = sw ? we_edge : (we_edge & (x[1][1:0]==2'b00));

    always @(posedge clk)
      if ( ntsc_we )
        begin
         ntsc_addr <= sw ? myaddr2 : myaddr;     // normal and expanded modes
         ntsc_data <= sw ? mydata2 : mydata;
        end

endmodule // ntsc_to_zbt
```

Appendix A: Video Tracking and Beat Detection: rgb_to_hsv.v

```
module rgb_to_hsv(r, g, b, clk, reset, h, s, v, ans, done);
input [7:0] r,g,b;
input clk,reset;
output [7:0] h,s,v;
output [13:0] ans;
output done;

wire [7:0] max, min;
wire [13:0] ans;
wire done;
wire [13:0] answer;
reg [7:0] h,num;
reg start;

assign v = max;
assign s = 8'b0;

always @ (posedge clk) begin
if (reset) begin
h <= 8'b0;
num <= 8'b0;
start <= 1'b0;
end
if (max == r) begin num <= (g >= b) ? g-b : b-g;
                                          h <= ((g >= b) && done) ? ans : 256-ans;
                                          start <= 1'b1; end
if (max == g) begin num <= (b >= r) ? b-r : r-b;
                                          h <= (( b>= r) && done) ? ans + 85 : 85 -
ans;

      start <= 1'b1; end
if (max == b) begin num <= (r >= g) ? r-g : g-r;
                                          h <= (( r>= g) && done) ? ans + 171 : 171
- ans;

      start <= 1'b1; end
if (max == min) h <= 8'b0;
end

assign ans = done ? answer : ans;
//module max_min_finder(clk, reset, r, g, b, max, min);
max_min_finder finder(clk, reset, r,g,b,max,min);

//module operation_divider(clk, reset, numerator, denom, ans, done);
//operation_divider divider360(clk, reset, 43*num, (max-min), ans, done);
streamlined_divider divider(answer,remainder,done,43*num,(max-min),start,clk);
endmodule
```

Appendix A: Video Tracking and Beat Detection: RGB_value_finder.v

```
module RGB_value_finder(clk, reset, hcount, vcount, x, y, R, G, B,
                                                red_coord, green_coord,
blue_coord);

input clk, reset;
input [10:0] hcount, x;
input [9:0] vcount, y;
input [7:0] R, G, B;
output [7:0] red_coord, green_coord, blue_coord;

reg [7:0] red_coord, green_coord, blue_coord;

always @ (posedge clk) begin
if (reset) begin
red_coord <= 8'b1;
green_coord <= 8'b1;
blue_coord <= 8'b1;
end
      if ((hcount == x) && (vcount == y)) begin
            red_coord <= R;
            green_coord <= G;
            blue_coord <= B;
            end
            else begin
            red_coord <= red_coord;
            green_coord <= green_coord;
            blue_coord <= blue_coord;
            end
end
endmodule
```

Appendix A: Video Tracking and Beat Detection: Threshold finder.v

```
module threshold_finder(clk, reset, hcount, vcount, red, green, blue, new_red,
                                                       new_green, new_blue,
hcount_avg, vcount_avg);

input [7:0] red, green, blue;
input clk,reset;
input [10:0] hcount;
input [9:0] vcount;
output [10:0] hcount_avg;
output [9:0] vcount_avg;
output [7:0] new_red, new_green, new_blue;

parameter red_max = 8'd255;//255
parameter red_min = 8'd150;//150
parameter green_max = 8'd116; //116
parameter green_min = 8'd30; //30
parameter blue_max = 8'd252; //252
parameter blue_min = 8'd110; //110

reg [7:0] new_red, new_green, new_blue;
reg [2:0] count; //max = 7
reg [22:0] hsum, vsum, hsum1, vsum1;
reg start;
reg [12:0] division_counter, division_counter1;


wire [15:0] hquotient;
wire hready;
wire [15:0] vquotient;
wire vready;
```

```
always @ (posedge clk) begin
if (reset) begin
new_red <= 8'b0;
new_green <= 8'b0;
new_blue <= 8'b0;
count <= 3'd0;
hsum1 <= 23'b0;
vsum1 <= 23'b0;
division_counter1 <= 13'd0;
end

if ((red >= red_min) && (red <= red_max) &&
            (green <= green_max) && (green >= green_min) &&
            (blue <= blue_max) && (blue >= blue_min) && (hcount < 11'd512))
begin // threshold
            if (count == 3'd7) begin
            new_red <= red;
            new_blue <= blue;
            new_green <= green;
            count <= count;

            hsum1 <= hsum1 + hcount;
            vsum1 <= vsum1 + vcount;
            division_counter1 <= division_counter1 + 1;
            end
            if (count < 3'd7) begin // get rid of the speckles
            new_red <= 8'b0;
            new_blue <= 8'b0;
            new_green <= 8'b0;
            count <= count + 1;
            end
            end
            else begin
            new_red <= 8'b0;
            new_blue <= 8'b0;
            new_green <= 8'b0;
            count <= 3'd0;
            end

if ((hcount == 1024) && (vcount == 768)) begin
hsum <= hsum1;
vsum <= vsum1;
division_counter <= (division_counter1 > 30) ? division_counter1 : 13'b0;

end
if ((hcount == 0) && (vcount == 0)) begin
hsum1 <= 23'b0;
vsum1 <= 23'b0;
division_counter1 <= 13'b0;
count <= 3'b0;
end
end

assign hcount_avg = hready ? hquotient[10:0] : hcount_avg;
assign vcount_avg = vready ? vquotient[9:0] : vcount_avg;

divider
```

```
divider1(.dividend(hsum), .divisor(division_counter), .quotient(hquotient), .clk(cl
k), .remainder(hremain), .rfd(hready));
divider

divider2(.dividend(vsum), .divisor(division_counter), .quotient(vquotient), .clk(cl
k), .remainder(vremain), .rfd(vready));

endmodule
```

Appendix A: Video Tracking and Beat Detection: video_decoder.v

```
//
// File:   video_decoder.v
// Date:   31-Oct-05
// Author: J. Castro (MIT 6.111, fall 2005)
//
// This file contains the ntsc_decode and adv7185init modules
//
// These modules are used to grab input NTSC video data from the RCA
// phono jack on the right hand side of the 6.111 labkit (connect
// the camera to the LOWER jack).
//

/////////////////////////////////////////////////////////////////////////////
//
// NTSC decode - 16-bit CCIR656 decoder
// By Javier Castro
// This module takes a stream of LLC data from the adv7185
// NTSC/PAL video decoder and generates the corresponding pixels,
// that are encoded within the stream, in YCrCb format.

// Make sure that the adv7185 is set to run in 16-bit LLC2 mode.

module ntsc_decode(clk, reset, tv_in_ycrcb, ycrcb, f, v, h, data_valid);

   // clk - line-locked clock (in this case, LLC1 which runs at 27Mhz)
   // reset - system reset
   // tv_in_ycrcb - 10-bit input from chip. should map to pins [19:10]
   // ycrcb - 24 bit luminance and chrominance (8 bits each)
   // f - field: 1 indicates an even field, 0 an odd field
   // v - vertical sync: 1 means vertical sync
   // h - horizontal sync: 1 means horizontal sync

   input clk;
   input reset;
   input [9:0] tv_in_ycrcb; // modified for 10 bit input - should be P[19:10]
   output [29:0] ycrcb;
   output   f;
   output   v;
   output   h;
   output   data_valid;
   // output [4:0] state;

   parameter      SYNC_1 = 0;
   parameter      SYNC_2 = 1;
   parameter      SYNC_3 = 2;
   parameter      SAV_f1_cb0 = 3;
   parameter      SAV_f1_y0 = 4;
   parameter      SAV_f1_cr1 = 5;
   parameter      SAV_f1_y1 = 6;
   parameter      EAV_f1 = 7;
   parameter      SAV_VBI_f1 = 8;
   parameter      EAV_VBI_f1 = 9;
   parameter      SAV_f2_cb0 = 10;
   parameter      SAV_f2_y0 = 11;
   parameter      SAV_f2_cr1 = 12;
   parameter      SAV_f2_y1 = 13;
   parameter      EAV_f2 = 14;
   parameter      SAV_VBI_f2 = 15;
   parameter      EAV_VBI_f2 = 16;




   // In the start state, the module doesn't know where
   // in the sequence of pixels, it is looking.

   // Once we determine where to start, the FSM goes through a normal
   // sequence of SAV process_YCrCb EAV... repeat

   // The data stream looks as follows
   // SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... |
EAV sequence
   // There are two things we need to do:
   //   1. Find the two SAV blocks (stands for Start Active Video perhaps?)
   //   2. Decode the subsequent data

   reg [4:0]      current_state = 5'h00;
   reg [9:0]      y = 10'h000;  // luminance
   reg [9:0]      cr = 10'h000; // chrominance
   reg [9:0]      cb = 10'h000; // more chrominance

   assign   state = current_state;

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

        end
      else
        begin
           // these states don't do much except allow us to know where we are in
the stream.
           // whenever the synchronization code is seen, go back to the sync_state
before
           // transitioning to the new state
           case (current_state)
             SYNC_1: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_2 : SYNC_1;
             SYNC_2: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_3 : SYNC_1;
             SYNC_3: current_state <= (tv_in_ycrcb == 10'h200) ? SAV_f1_cb0 :
                              (tv_in_ycrcb == 10'h274) ? EAV_f1 :
                              (tv_in_ycrcb == 10'h2ac) ? SAV_VBI_f1 :
                              (tv_in_ycrcb == 10'h2d8) ? EAV_VBI_f1 :
```

```
                              (tv_in_ycrcb == 10'h31c) ? SAV_f2_cb0 :
                              (tv_in_ycrcb == 10'h368) ? EAV_f2 :
                              (tv_in_ycrcb == 10'h3b0) ? SAV_VBI_f2 :
                              (tv_in_ycrcb == 10'h3c4) ? EAV_VBI_f2 : SYNC_1;

             SAV_f1_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 :
SAV_f1_y0;
             SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 :
SAV_f1_cr1;
             SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 :
SAV_f1_y1;
             SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 :
SAV_f1_cb0;

             SAV_f2_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 :
SAV_f2_y0;
             SAV_f2_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 :
SAV_f2_cr1;
             SAV_f2_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 :
SAV_f2_y1;
             SAV_f2_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 :
SAV_f2_cb0;

             // These states are here in the event that we want to cover these
signals
             // in the future. For now, they just send the state machine back to
SYNC_1
             EAV_f1: current_state <= SYNC_1;
             SAV_VBI_f1: current_state <= SYNC_1;
             EAV_VBI_f1: current_state <= SYNC_1;
             EAV_f2: current_state <= SYNC_1;
             SAV_VBI_f2: current_state <= SYNC_1;
             EAV_VBI_f2: current_state <= SYNC_1;

           endcase
        end
     end // always @ (posedge clk)

   // implement our decoding mechanism

   wire y_enable;
   wire cr_enable;
   wire cb_enable;

   // if y is coming in, enable the register
   // likewise for cr and cb
   assign y_enable = (current_state == SAV_f1_y0) ||
                    (current_state == SAV_f1_y1) ||
                    (current_state == SAV_f2_y0) ||
                    (current_state == SAV_f2_y1);
   assign cr_enable = (current_state == SAV_f1_cr1) ||
                     (current_state == SAV_f2_cr1);
   assign cb_enable = (current_state == SAV_f1_cb0) ||
                     (current_state == SAV_f2_cb0);

   // f, v, and h only go high when active
   assign {v,h} = (current_state == SYNC_3) ? tv_in_ycrcb[7:6] : 2'b00;
   // data is valid when we have all three values: y, cr, cb
   assign data_valid = y_enable;
   assign ycrcb = {y,cr,cb};

   reg        f = 0;

   always @ (posedge clk)
     begin
      y <= y_enable ? tv_in_ycrcb : y;
      cr <= cr_enable ? tv_in_ycrcb : cr;
      cb <= cb_enable ? tv_in_ycrcb : cb;
      f <= (current_state == SYNC_3) ? tv_in_ycrcb[8] : f;
     end

endmodule



///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ADV7185 Video Decoder Configuration Init
//
// Created:
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
// Register 0
///////////////////////////////////////////////////////////////////////////////

`define INPUT_SELECT                            4'h0
  // 0: CVBS on AIN1 (composite video in)
  // 7: Y on AIN2, C on AIN5 (s-video in)
  // (These are the only configurations supported by the 6.111 labkit hardware)
`define INPUT_MODE                              4'h0
  // 0: Autodetect: NTSC or PAL (BGHID), w/o pedestal
  // 1: Autodetect: NTSC or PAL (BGHID), w/pedestal
  // 2: Autodetect: NTSC or PAL (N), w/o pedestal
  // 3: Autodetect: NTSC or PAL (N), w/pedestal
  // 4: NTSC w/o pedestal
  // 5: NTSC w/pedestal
  // 6: NTSC 4.43 w/o pedestal
  // 7: NTSC 4.43 w/pedestal
  // 8: PAL BGHID w/o pedestal
  // 9: PAL N w/pedestal
  // A: PAL M w/o pedestal
  // B: PAL M w/pedestal
  // C: PAL combination N
  // D: PAL combination N w/pedestal
  // E-F: [Not valid]

`define ADV7185_REGISTER_0 {`INPUT_MODE, `INPUT_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 1
```

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

`define VIDEO_QUALITY                          2'h0
  // 0: Broadcast quality
  // 1: TV quality
  // 2: VCR quality
  // 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE                   1'b0
  // 0: Normal mode
  // 1: Square pixel mode
`define DIFFERENTIAL_INPUT                     1'b0
  // 0: Single-ended inputs
  // 1: Differential inputs
`define FOUR_TIMES_SAMPLING                    1'b0
  // 0: Standard sampling rate
  // 1: 4x sampling rate (NTSC only)
`define BETACAM                                1'b0
  // 0: Standard video input
  // 1: Betacam video input
`define AUTOMATIC_STARTUP_ENABLE               1'b1
  // 0: Change of input triggers reacquire
  // 1: Change of input does not trigger reacquire

`define ADV7185_REGISTER_1 {`AUTOMATIC_STARTUP_ENABLE, 1'b0, `BETACAM,
`FOUR_TIMES_SAMPLING, `DIFFERENTIAL_INPUT, `SQUARE_PIXEL_IN_MODE, `VIDEO_QUALITY}

///////////////////////////////////////////////////////////////////////////////
// Register 2
///////////////////////////////////////////////////////////////////////////////

`define Y_PEAKING_FILTER                       3'h4
  // 0: Composite =  4.5dB,  s-video =  9.25dB
  // 1: Composite =  4.5dB,  s-video =  9.25dB
  // 2: Composite =  4.5dB,  s-video =  5.75dB
  // 3: Composite =  1.25dB, s-video =  3.3dB
  // 4: Composite =  0.0dB,  s-video =  0.0dB
  // 5: Composite = -1.25dB, s-video = -3.0dB
  // 6: Composite = -1.75dB, s-video = -8.0dB
  // 7: Composite = -3.0dB,  s-video = -8.0dB
`define CORING                                 2'h0
  // 0: No coring
  // 1: Truncate if Y < black+8
  // 2: Truncate if Y < black+16
  // 3: Truncate if Y < black+32

`define ADV7185_REGISTER_2 {3'b000, `CORING, `Y_PEAKING_FILTER}

///////////////////////////////////////////////////////////////////////////////
// Register 3
///////////////////////////////////////////////////////////////////////////////

`define INTERFACE_SELECT                       2'h0
  // 0: Philips-compatible
  // 1: Broktree API A-compatible
  // 2: Broktree API B-compatible
  // 3: [Not valid]
`define OUTPUT_FORMAT                          4'h0
```

```
  // 0: 10-bit @ LLC, 4:2:2 CCIR656
  // 1: 20-bit @ LLC, 4:2:2 CCIR656
  // 2: 16-bit @ LLC, 4:2:2 CCIR656
  // 3: 8-bit @ LLC, 4:2:2 CCIR656
  // 4: 12-bit @ LLC, 4:1:1
  // 5-F: [Not valid]
  // (Note that the 6.111 labkit hardware provides only a 10-bit interface to
  // the ADV7185.)
`define TRISTATE_OUTPUT_DRIVERS                  1'b0
  // 0: Drivers tristated when ~OE is high
  // 1: Drivers always tristated
`define VBI_ENABLE                               1'b0
  // 0: Decode lines during vertical blanking interval
  // 1: Decode only active video regions

`define ADV7185_REGISTER_3 {`VBI_ENABLE, `TRISTATE_OUTPUT_DRIVERS, `OUTPUT_FORMAT,
`INTERFACE_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 4
///////////////////////////////////////////////////////////////////////////////

`define OUTPUT_DATA_RANGE                        1'b0
  // 0: Output values restricted to CCIR-compliant range
  // 1: Use full output range
`define BT656_TYPE                               1'b0
  // 0: BT656-3-compatible
  // 1: BT656-4-compatible

`define ADV7185_REGISTER_4 {`BT656_TYPE, 3'b000, 3'b110, `OUTPUT_DATA_RANGE}

///////////////////////////////////////////////////////////////////////////////
// Register 5
///////////////////////////////////////////////////////////////////////////////


`define GENERAL_PURPOSE_OUTPUTS                  4'b0000
`define GPO_0_1_ENABLE                           1'b0
  // 0: General purpose outputs 0 and 1 tristated
  // 1: General purpose outputs 0 and 1 enabled
`define GPO_2_3_ENABLE                           1'b0
  // 0: General purpose outputs 2 and 3 tristated
  // 1: General purpose outputs 2 and 3 enabled
`define BLANK_CHROMA_IN_VBI                      1'b1
  // 0: Chroma decoded and output during vertical blanking
  // 1: Chroma blanked during vertical blanking
`define HLOCK_ENABLE                             1'b0
  // 0: GPO 0 is a general purpose output
  // 1: GPO 0 shows HLOCK status

`define ADV7185_REGISTER_5 {`HLOCK_ENABLE, `BLANK_CHROMA_IN_VBI, `GPO_2_3_ENABLE,
`GPO_0_1_ENABLE, `GENERAL_PURPOSE_OUTPUTS}

///////////////////////////////////////////////////////////////////////////////
// Register 7
///////////////////////////////////////////////////////////////////////////////
```

```
`define FIFO_FLAG_MARGIN                        5'h10
  // Sets the locations where FIFO almost-full and almost-empty flags are set
`define FIFO_RESET                              1'b0
  // 0: Normal operation
  // 1: Reset FIFO. This bit is automatically cleared
`define AUTOMATIC_FIFO_RESET                    1'b0
  // 0: No automatic reset
  // 1: FIFO is autmatically reset at the end of each video field
`define FIFO_FLAG_SELF_TIME                     1'b1
  // 0: FIFO flags are synchronized to CLKIN
  // 1: FIFO flags are synchronized to internal 27MHz clock

`define ADV7185_REGISTER_7 {`FIFO_FLAG_SELF_TIME, `AUTOMATIC_FIFO_RESET,
`FIFO_RESET, `FIFO_FLAG_MARGIN}

///////////////////////////////////////////////////////////////////////////////
// Register 8
///////////////////////////////////////////////////////////////////////////////

`define INPUT_CONTRAST_ADJUST                           8'h80

`define ADV7185_REGISTER_8 {`INPUT_CONTRAST_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register 9
///////////////////////////////////////////////////////////////////////////////

`define INPUT_SATURATION_ADJUST                         8'h8C

`define ADV7185_REGISTER_9 {`INPUT_SATURATION_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register A
///////////////////////////////////////////////////////////////////////////////

`define INPUT_BRIGHTNESS_ADJUST                         8'h00

`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register B
///////////////////////////////////////////////////////////////////////////////

`define INPUT_HUE_ADJUST                                8'h00

`define ADV7185_REGISTER_B {`INPUT_HUE_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register C
///////////////////////////////////////////////////////////////////////////////

`define DEFAULT_VALUE_ENABLE                    1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values
`define DEFAULT_VALUE_AUTOMATIC_ENABLE          1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values if lock is lost
```

```
`define DEFAULT_Y_VALUE                        6'h0C
  // Default Y value

`define ADV7185_REGISTER_C {`DEFAULT_Y_VALUE, `DEFAULT_VALUE_AUTOMATIC_ENABLE,
`DEFAULT_VALUE_ENABLE}

///////////////////////////////////////////////////////////////////////////////
// Register D
///////////////////////////////////////////////////////////////////////////////

`define DEFAULT_CR_VALUE                       4'h8
  // Most-significant four bits of default Cr value
`define DEFAULT_CB_VALUE                       4'h8
  // Most-significant four bits of default Cb value

`define ADV7185_REGISTER_D {`DEFAULT_CB_VALUE, `DEFAULT_CR_VALUE}

///////////////////////////////////////////////////////////////////////////////
// Register E
///////////////////////////////////////////////////////////////////////////////

`define TEMPORAL_DECIMATION_ENABLE             1'b0
  // 0: Disable
  // 1: Enable
`define TEMPORAL_DECIMATION_CONTROL            2'h0
  // 0: Supress frames, start with even field
  // 1: Supress frames, start with odd field
  // 2: Supress even fields only
  // 3: Supress odd fields only
`define TEMPORAL_DECIMATION_RATE               4'h0
  // 0-F: Number of fields/frames to skip

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE,
`TEMPORAL_DECIMATION_CONTROL, `TEMPORAL_DECIMATION_ENABLE}

///////////////////////////////////////////////////////////////////////////////
// Register F
///////////////////////////////////////////////////////////////////////////////

`define POWER_SAVE_CONTROL                     2'h0
  // 0: Full operation
  // 1: CVBS only
  // 2: Digital only
  // 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY             1'b0
  // 0: Power-down pin has priority
  // 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE                   1'b0
  // 0: Reference is functional
  // 1: Reference is powered down
`define POWER_DOWN_LLC_GENERATOR               1'b0
  // 0: LLC generator is functional
  // 1: LLC generator is powered down
`define POWER_DOWN_CHIP                        1'b0
  // 0: Chip is functional
  // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE                       1'b0
```

```
  // 0: Normal operation
  // 1: Reacquire video signal (bit will automatically reset)
`define RESET_CHIP                                1'b0
  // 0: Normal operation
  // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP,
`POWER_DOWN_LLC_GENERATOR, `POWER_DOWN_REFERENCE, `POWER_DOWN_SOURCE_PRIORITY,
`POWER_SAVE_CONTROL}

////////////////////////////////////////////////////////////////////////////////
// Register 33
////////////////////////////////////////////////////////////////////////////////

`define PEAK_WHITE_UPDATE                         1'b1
  // 0: Update gain once per line
  // 1: Update gain once per field
`define AVERAGE_BIRIGHTNESS_LINES                 1'b1
  // 0: Use lines 33 to 310
  // 1: Use lines 33 to 270
`define MAXIMUM_IRE                               3'h0
  // 0: PAL: 133, NTSC: 122
  // 1: PAL: 125, NTSC: 115
  // 2: PAL: 120, NTSC: 110
  // 3: PAL: 115, NTSC: 105
  // 4: PAL: 110, NTSC: 100
  // 5: PAL: 105, NTSC: 100
  // 6-7: PAL: 100, NTSC: 100
`define COLOR_KILL                                1'b1
  // 0: Disable color kill
  // 1: Enable color kill

`define ADV7185_REGISTER_33 {1'b1, `COLOR_KILL, 1'b1, `MAXIMUM_IRE,
`AVERAGE_BIRIGHTNESS_LINES, `PEAK_WHITE_UPDATE}

`define ADV7185_REGISTER_10 8'h00
`define ADV7185_REGISTER_11 8'h00
`define ADV7185_REGISTER_12 8'h00
`define ADV7185_REGISTER_13 8'h45
`define ADV7185_REGISTER_14 8'h18
`define ADV7185_REGISTER_15 8'h60
`define ADV7185_REGISTER_16 8'h00
`define ADV7185_REGISTER_17 8'h01
`define ADV7185_REGISTER_18 8'h00
`define ADV7185_REGISTER_19 8'h10
`define ADV7185_REGISTER_1A 8'h10
`define ADV7185_REGISTER_1B 8'hF0
`define ADV7185_REGISTER_1C 8'h16
`define ADV7185_REGISTER_1D 8'h01
`define ADV7185_REGISTER_1E 8'h00
`define ADV7185_REGISTER_1F 8'h3D
`define ADV7185_REGISTER_20 8'hD0
`define ADV7185_REGISTER_21 8'h09
`define ADV7185_REGISTER_22 8'h8C
`define ADV7185_REGISTER_23 8'hE2
`define ADV7185_REGISTER_24 8'h1F
`define ADV7185_REGISTER_25 8'h07
```

```
`define ADV7185_REGISTER_26 8'hC2
`define ADV7185_REGISTER_27 8'h58
`define ADV7185_REGISTER_28 8'h3C
`define ADV7185_REGISTER_29 8'h00
`define ADV7185_REGISTER_2A 8'h00
`define ADV7185_REGISTER_2B 8'hA0
`define ADV7185_REGISTER_2C 8'hCE
`define ADV7185_REGISTER_2D 8'hF0
`define ADV7185_REGISTER_2E 8'h00
`define ADV7185_REGISTER_2F 8'hF0
`define ADV7185_REGISTER_30 8'h00
`define ADV7185_REGISTER_31 8'h70
`define ADV7185_REGISTER_32 8'h00
`define ADV7185_REGISTER_34 8'h0F
`define ADV7185_REGISTER_35 8'h01
`define ADV7185_REGISTER_36 8'h00
`define ADV7185_REGISTER_37 8'h00
`define ADV7185_REGISTER_38 8'h00
`define ADV7185_REGISTER_39 8'h00
`define ADV7185_REGISTER_3A 8'h00
`define ADV7185_REGISTER_3B 8'h00

`define ADV7185_REGISTER_44 8'h41
`define ADV7185_REGISTER_45 8'hBB

`define ADV7185_REGISTER_F1 8'hEF
`define ADV7185_REGISTER_F2 8'h80


module adv7185init (reset, clock_27mhz, source, tv_in_reset_b,
                tv_in_i2c_clock, tv_in_i2c_data);

   input reset;
   input clock_27mhz;
   output tv_in_reset_b; // Reset signal to ADV7185
   output tv_in_i2c_clock; // I2C clock output to ADV7185
   output tv_in_i2c_data; // I2C data line to ADV7185
   input source; // 0: composite, 1: s-video

   initial begin
      $display("ADV7185 Initialization values:");
      $display("  Register 0:  0x%X", `ADV7185_REGISTER_0);
      $display("  Register 1:  0x%X", `ADV7185_REGISTER_1);
      $display("  Register 2:  0x%X", `ADV7185_REGISTER_2);
      $display("  Register 3:  0x%X", `ADV7185_REGISTER_3);
      $display("  Register 4:  0x%X", `ADV7185_REGISTER_4);
      $display("  Register 5:  0x%X", `ADV7185_REGISTER_5);
      $display("  Register 7:  0x%X", `ADV7185_REGISTER_7);
      $display("  Register 8:  0x%X", `ADV7185_REGISTER_8);
      $display("  Register 9:  0x%X", `ADV7185_REGISTER_9);
      $display("  Register A:  0x%X", `ADV7185_REGISTER_A);
      $display("  Register B:  0x%X", `ADV7185_REGISTER_B);
      $display("  Register C:  0x%X", `ADV7185_REGISTER_C);
      $display("  Register D:  0x%X", `ADV7185_REGISTER_D);
      $display("  Register E:  0x%X", `ADV7185_REGISTER_E);
      $display("  Register F:  0x%X", `ADV7185_REGISTER_F);
      $display("  Register 33: 0x%X", `ADV7185_REGISTER_33);
```

```
end

//
// Generate a 1MHz for the I2C driver (resulting I2C clock rate is 250kHz)
//

reg [7:0] clk_div_count, reset_count;
reg clock_slow;
wire reset_slow;

initial
  begin
   clk_div_count <= 8'h00;
   // synthesis attribute init of clk_div_count is "00";
   clock_slow <= 1'b0;
   // synthesis attribute init of clock_slow is "0";
  end

always @(posedge clock_27mhz)
  if (clk_div_count == 26)
    begin
     clock_slow <= ~clock_slow;
     clk_div_count <= 0;
    end
  else
    clk_div_count <= clk_div_count+1;

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

assign reset_slow = reset_count != 0;

//
// I2C driver
//

reg load;
reg [7:0] data;
wire ack, idle;

i2c i2c(.reset(reset_slow), .clock4x(clock_slow), .data(data), .load(load),
      .ack(ack), .idle(idle), .scl(tv_in_i2c_clock),
      .sda(tv_in_i2c_data));

//
// State machine
//

reg [7:0] state;
reg tv_in_reset_b;
reg old_source;

always @(posedge clock_slow)
   if (reset_slow)
```

```
begin
   state <= 0;
   load <= 0;
   tv_in_reset_b <= 0;
   old_source <= 0;
end
else
case (state)
  8'h00:
    begin
       // Assert reset
       load <= 1'b0;
       tv_in_reset_b <= 1'b0;
       if (!ack)
       state <= state+1;
    end
  8'h01:
    state <= state+1;
  8'h02:
    begin
       // Release reset
       tv_in_reset_b <= 1'b1;
       state <= state+1;
                end
  8'h03:
    begin
       // Send ADV7185 address
       data <= 8'h8A;
       load <= 1'b1;
       if (ack)
       state <= state+1;
    end
  8'h04:
    begin
       // Send subaddress of first register
       data <= 8'h00;
       if (ack)
       state <= state+1;
    end
  8'h05:
    begin
       // Write to register 0
       data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
       if (ack)
       state <= state+1;
    end
  8'h06:
    begin
       // Write to register 1
       data <= `ADV7185_REGISTER_1;
       if (ack)
       state <= state+1;
    end
  8'h07:
    begin
       // Write to register 2
       data <= `ADV7185_REGISTER_2;
```

```
     if (ack)
     state <= state+1;
  end
8'h08:
  begin
     // Write to register 3
     data <= `ADV7185_REGISTER_3;
     if (ack)
     state <= state+1;
  end
8'h09:
  begin
     // Write to register 4
     data <= `ADV7185_REGISTER_4;
     if (ack)
     state <= state+1;
  end
8'h0A:
  begin
     // Write to register 5
     data <= `ADV7185_REGISTER_5;
     if (ack)
     state <= state+1;
  end
8'h0B:
  begin
     // Write to register 6
     data <= 8'h00; // Reserved register, write all zeros
     if (ack)
     state <= state+1;
  end
8'h0C:
  begin
     // Write to register 7
     data <= `ADV7185_REGISTER_7;
     if (ack)
     state <= state+1;
  end
8'h0D:
  begin
     // Write to register 8
     data <= `ADV7185_REGISTER_8;
     if (ack)
     state <= state+1;
  end
8'h0E:
  begin
     // Write to register 9
     data <= `ADV7185_REGISTER_9;
     if (ack)
     state <= state+1;
  end
8'h0F: begin
    // Write to register A
    data <= `ADV7185_REGISTER_A;
 if (ack)
    state <= state+1;
```

```
end
8'h10:
  begin
     // Write to register B
     data <= `ADV7185_REGISTER_B;
     if (ack)
     state <= state+1;
  end
8'h11:
  begin
     // Write to register C
     data <= `ADV7185_REGISTER_C;
     if (ack)
     state <= state+1;
  end
8'h12:
  begin
     // Write to register D
     data <= `ADV7185_REGISTER_D;
     if (ack)
     state <= state+1;
  end
8'h13:
  begin
     // Write to register E
     data <= `ADV7185_REGISTER_E;
     if (ack)
     state <= state+1;
  end
8'h14:
  begin
     // Write to register F
     data <= `ADV7185_REGISTER_F;
     if (ack)
     state <= state+1;
  end
8'h15:
  begin
     // Wait for I2C transmitter to finish
     load <= 1'b0;
     if (idle)
     state <= state+1;
  end
8'h16:
  begin
     // Write address
     data <= 8'h8A;
     load <= 1'b1;
     if (ack)
     state <= state+1;
  end
8'h17:
  begin
     data <= 8'h33;
     if (ack)
     state <= state+1;
  end
```

```
8'h18:
  begin
     data <= `ADV7185_REGISTER_33;
     if (ack)
     state <= state+1;
  end
8'h19:
  begin
     load <= 1'b0;
     if (idle)
     state <= state+1;
  end

8'h1A: begin
   data <= 8'h8A;
   load <= 1'b1;
   if (ack)
     state <= state+1;
end
8'h1B:
  begin
     data <= 8'h33;
     if (ack)
     state <= state+1;
  end
8'h1C:
  begin
     load <= 1'b0;
     if (idle)
     state <= state+1;
  end
8'h1D:
  begin
     load <= 1'b1;
     data <= 8'h8B;
     if (ack)
     state <= state+1;
  end
8'h1E:
  begin
     data <= 8'hFF;
     if (ack)
     state <= state+1;
  end
8'h1F:
  begin
     load <= 1'b0;
     if (idle)
     state <= state+1;
  end
8'h20:
  begin
     // Idle
     if (old_source != source) state <= state+1;
     old_source <= source;
  end
8'h21: begin
```

```
            // Send ADV7185 address
            data <= 8'h8A;
            load <= 1'b1;
            if (ack) state <= state+1;
         end
         8'h22: begin
            // Send subaddress of register 0
            data <= 8'h00;
            if (ack) state <= state+1;
         end
         8'h23: begin
            // Write to register 0
            data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
            if (ack) state <= state+1;
         end
         8'h24: begin
            // Wait for I2C transmitter to finish
            load <= 1'b0;
            if (idle) state <= 8'h20;
         end
        endcase

endmodule

// i2c module for use with the ADV7185

module i2c (reset, clock4x, data, load, idle, ack, scl, sda);

   input reset;
   input clock4x;
   input [7:0] data;
   input load;
   output ack;
   output idle;
   output scl;
   output sda;

   reg [7:0] ldata;
   reg ack, idle;
   reg scl;
   reg sdai;

   reg [7:0] state;

   assign sda = sdai ? 1'bZ : 1'b0;

   always @(posedge clock4x)
     if (reset)
       begin
        state <= 0;
        ack <= 0;
       end
     else
       case (state)
       8'h00: // idle
         begin
            scl <= 1'b1;
            sdai <= 1'b1;
            ack <= 1'b0;
            idle <= 1'b1;
            if (load)
            begin
               ldata <= data;
               ack <= 1'b1;
               state <= state+1;
            end
         end
       8'h01: // Start
         begin
            ack <= 1'b0;
            idle <= 1'b0;
            sdai <= 1'b0;
            state <= state+1;
         end
       8'h02:
         begin
            scl <= 1'b0;
            state <= state+1;
         end
       8'h03: // Send bit 7
         begin
            ack <= 1'b0;
            sdai <= ldata[7];
            state <= state+1;
         end
       8'h04:
         begin
            scl <= 1'b1;
            state <= state+1;
         end
       8'h05:
         begin
            state <= state+1;
         end
       8'h06:
         begin
            scl <= 1'b0;
            state <= state+1;
         end
       8'h07:
         begin
            sdai <= ldata[6];
            state <= state+1;
         end
       8'h08:
         begin
            scl <= 1'b1;
            state <= state+1;
         end
       8'h09:
         begin
            state <= state+1;
         end
       8'h0A:
```

```
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h0B:
  begin
     sdai <= ldata[5];
     state <= state+1;
  end
8'h0C:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h0D:
  begin
     state <= state+1;
  end
8'h0E:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h0F:
  begin
     sdai <= ldata[4];
     state <= state+1;
  end
8'h10:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h11:
  begin
     state <= state+1;
  end
8'h12:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h13:
  begin
     sdai <= ldata[3];
     state <= state+1;
  end
8'h14:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h15:
  begin
     state <= state+1;
  end
8'h16:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h17:
  begin
     sdai <= ldata[2];
     state <= state+1;
  end
8'h18:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h19:
  begin
     state <= state+1;
  end
8'h1A:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1B:
  begin
     sdai <= ldata[1];
     state <= state+1;
  end
8'h1C:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h1D:
  begin
     state <= state+1;
  end
8'h1E:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1F:
  begin
     sdai <= ldata[0];
     state <= state+1;
  end
8'h20:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h21:
  begin
     state <= state+1;
  end
8'h22:
        begin
           scl <= 1'b0;
           state <= state+1;
        end
      8'h23: // Acknowledge bit
        begin
           state <= state+1;
        end
      8'h24:
        begin
           scl <= 1'b1;
           state <= state+1;
        end
      8'h25:
        begin
           state <= state+1;
        end
      8'h26:
        begin
           scl <= 1'b0;
           if (load)
           begin
              ldata <= data;
              ack <= 1'b1;
              state <= 3;
           end
           else
           state <= state+1;
        end
      8'h27:
        begin
           sdai <= 1'b0;
           state <= state+1;
        end
      8'h28:
        begin
           scl <= 1'b1;
           state <= state+1;
        end
      8'h29:
        begin
           sdai <= 1'b1;
           state <= 0;
        end
      endcase
endmodule
```

Appendix A: Video Tracking and Beat Detection: volume_control.v

```
module volume_control(clk, reset, noisy, clean);
input clk, reset;
input [3:0] noisy;
output [3:0] clean;

reg [3:0] clean;
reg [3:0] counter;
always @ (posedge clk) begin
```

```
if (reset) begin
counter <= 4'b0;
clean <= noisy;
end

if (counter == 15) begin
      counter <= 4'b0;
      if (clean > noisy) begin
                  clean <= (clean == 0) ? clean : clean - 1;
                        end
      if (clean == noisy) clean <= noisy;
      if (clean < noisy) begin
            clean <= (clean == 16) ? clean : clean + 1;
            end
end

else counter <= counter + 1;

end
endmodule
```

Appendix A: Video Tracking and Beat Detection: vram_display.v

```
//////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.

module vram_display(reset,clk,hcount,vcount,vr_pixel,
                vram_addr,vram_read_data);

   input reset, clk;
   input [10:0] hcount;
   input [9:0]    vcount;
   output [17:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

   wire [18:0]     vram_addr = {vcount, hcount[9:1]};

//parameter HMID = 9'd367; // The horizontal center of the image in MEMORY
//parameter HSTART = HMID-9'd256; // The horizontal counter decrements!!!
//parameter VMID = 9'd287; // The vertical center of the image in MEMORY
//parameter VSTART = VMID-9'd192;
//
//wire [18:0] vram_addr = {vcount+VSTART, ~hcount[9:1]-9'd180};
////wire [1:0] hc4 = hcount[1:0];


   wire      hc2 = hcount[0];
   reg [17:0]      vr_pixel;
   reg [35:0]      vr_data_latched;
   reg [35:0]      last_vr_data;
   always @(posedge clk)
     last_vr_data <= (hc2==1'd0) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (hc2==1'd1) ? vram_read_data : vr_data_latched;

   always @(*)          // each 36-bit word from RAM is decoded to 2 bytes
     case (hc2)
       2'd0: vr_pixel = last_vr_data[17:0];
       2'd1: vr_pixel = last_vr_data[35:18];

     endcase

endmodule // vram_display

/////////////////////////////////////////////////////////////////////////////
// parameterized delay line

module delayN(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 3;

   reg [NDELAY-1:0] shiftreg;
   wire         out = shiftreg[NDELAY-1];

   always @(posedge clk)
     shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule // delayN
```

Appendix A: Video Tracking and Beat Detection: zbt_6111.v

```
//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//

/////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.
```

```
module zbt_6111(clk, cen, we, addr, write_data, read_data,
              ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

   input clk;                 // system clock
   input cen;                 // clock enable for gating ZBT cycles
   input we;                  // write enable (active HIGH)
   input [18:0] addr;         // memory address
   input [35:0] write_data;   // data to write
   output [35:0] read_data;   // data read from memory
   output    ram_clk;   // physical line to ram clock
   output    ram_we_b;  // physical line to ram we_b
   output [18:0] ram_address; // physical line to ram address
   inout [35:0]  ram_data;    // physical line to ram data
   output    ram_cen_b; // physical line to ram clock enable

   // clock enable (should be synchronous and one cycle high at a time)
   wire      ram_cen_b = ~cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;

   always @(posedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;

   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

   // wire to ZBT RAM signals

   assign      ram_we_b = ~we;
   assign      ram_clk = ~clk;     // RAM is not happy with our data hold
                                   // times if its clk edges equal FPGA's
                                   // so we clock it on the falling edges
                                   // and thus let data stabilize longer
   assign      ram_address = addr;

   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
   assign      read_data = ram_data;

endmodule // zbt_6111
```

Appendix A: Video Tracking and Beat Detection: zbt_6111_sample.v

```
//
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT
// memories for video display.  Video input from the NTSC digitizer is
// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is used
// as the video frame buffer, with 8 bits used per pixel (black & white).
//
// Since the ZBT is read once for every four pixels, this frees up time for
// data to be stored to the ZBT during other pixel times.  The NTSC decoder
// runs at 27 MHz, whereas the XGA runs at 65 MHz, so we synchronize
// signals between the two (see ntsc2zbt.v) and let the NTSC data be
// stored to ZBT memory whenever it is available, during cycles when
// pixel reads are not being performed.
//
// We use a very simple ZBT interface, which does not involve any clock
// generation or hiding of the pipelining.  See zbt_6111.v for more info.
//
// switch[7] selects between display of NTSC video and test bars
// switch[6] is used for testing the NTSC decoder
// switch[1] selects between test bar periods; these are stored to ZBT
//           during blanking periods
// switch[0] selects vertical test bars (hardwired; not stored in ZBT)
//
//`include "display_16hex.v"
//`include "debounce.v"
//`include "video_decoder.v"
//`include "zbt_6111.v"
//`include "ntsc2zbt.v"

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

module zbt_6111_sample(beep, audio_reset_b,
                   ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock,

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

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

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

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

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

             clock_feedback_out, clock_feedback_in,

             flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
             flash_reset_b, flash_sts, flash_byte_b,
             rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

             mouse_clock, mouse_data, keyboard_clock, keyboard_data,

             clock_27mhz, clock1, clock2,

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

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

             switch,

             led,

             user1, user2, user3, user4,

             daughtercard,

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

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

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

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

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

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

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

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

```
   input  clock_feedback_in;
   output clock_feedback_out;

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

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

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

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

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

   inout [43:0] daughtercard;

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

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

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

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

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

   // Video Input
   //assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b1;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b1;
   //assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = clock_27mhz;//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

/* change lines below to enable ZBT RAM bank0 */

/*
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;  // clock enable
*/

/* enable RAM pins */

   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_bwe_b = 4'h0;

/**********/

   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1'b1;
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;

   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

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

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

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

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

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

   // User I/Os
   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;

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

   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(power_on_reset, clk, ~button_enter, user_reset);
   assign reset = user_reset | power_on_reset;

         reg [63:0] dispdatawire;
            wire [10:0] hcount_avg,  x_init, prev_x;
            wire [9:0] vcount_avg;
            wire [7:0] counter;
            wire beats;
            wire [3:0] beat_counter;
            wire east, ready, pulse, n_clock, west;
            wire [3:0] volume, volume_clean;
//          //module conductor(clock_27mhz, reset, tv_in_line_clock1, tv_in_ycrcb,
switch0, switch1, switch6, switch7, vga_out_red,
//                            vga_out_blue,  vga_out_green, beats, tv_in_reset_b,
vram_read_data, ram0_data, ram0_clk, ram0_we_b,
//                            ram0_address, ram0_cen_b, vga_out_sync_b,
vga_out_blank_b, vga_out_hsync, vga_out_vsync, vga_out_pixel_clock,
tv_in_i2c_clock, tv_in_i2c_data,
//                            counter, hcount_avg, vcount_avg, clk, east, ready,
pulse, n_clock);
conductor conductor1(clock_27mhz, reset, tv_in_line_clock1, tv_in_ycrcb, switch[0],
switch[1], switch[6], switch[7], vga_out_red,
                              vga_out_blue,  vga_out_green, beats, tv_in_reset_b,
vram_read_data, ram0_data, ram0_clk, ram0_we_b,
                              ram0_address, ram0_cen_b, vga_out_sync_b,
vga_out_blank_b, vga_out_hsync, vga_out_vsync, vga_out_pixel_clock,
tv_in_i2c_clock, tv_in_i2c_data,
                              counter, hcount_avg, vcount_avg, clk,  east, ready,
pulse, n_clock, x_init, prev_x, west, volume, volume_clean, beat_counter);


  always @(posedge clk)
     // dispdata <= {vram_read_data,9'b0,vram_addr};
//     dispdata <= {ntsc_data,9'b0,ntsc_addr}; // original code
            dispdatawire <= {volume, volume_clean, 13'b0, x_init , counter,
1'b0,hcount_avg, 2'b0, vcount_avg};
      assign analyzer1_data = {4'b0, volume_clean, 4'b0, volume};
   assign analyzer1_clock = n_clock;
      assign analyzer3_data = {3'b0, east, west, pulse, beats, 1'b0, counter};
   assign analyzer3_clock =  n_clock;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock =  n_clock;
   display_16hex hexdisp1(reset, clk, dispdatawire,
                    disp_blank, disp_clock, disp_rs, disp_ce_b,
                    disp_reset_b, disp_data_out);

   assign led = ~{1'b0, west,east,ready, pulse, n_clock, reset, beats};
```

```
endmodule
```

Appendix A: labkit.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green"
//              and "vga_out_blue". (Was 10'h0, now 8'h0.)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//               be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//               actually populated on the boards. (The boards support up to
//               72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

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

          clock_feedback_out, clock_feedback_in,

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

          rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

          mouse_clock, mouse_data, keyboard_clock, keyboard_data,

          clock_27mhz, clock1, clock2,

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

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

          switch,

          led,

          user1, user2, user3, user4,

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

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

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

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

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

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

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

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

   input  clock_feedback_in;
   output clock_feedback_out;

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

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

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

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

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

   inout [43:0] daughtercard;

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

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

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

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

   // VGA Output
   assign vga_out_red = 8'h0;
   assign vga_out_green = 8'h0;
   assign vga_out_blue = 8'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

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

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

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

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

   // User I/Os
   assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   assign user3[31:10] = 22'hZ;
   assign user4[31:5] = 27'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

   // 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



////////////////////////////////////////////////////
///User Input / Debug Output////////////////////////
////////////////////////////////////////////////////
   wire reset;
   debounce clean_reset(0, clock_27mhz, switch[0], reset);
   wire writemode;
   debounce clean_writemode(0, clock_27mhz, switch[1], writemode);

   wire beat_in_video,beat_in_switch,beat_in,which_beat,artificial;
   debounce clean_beatin(0, clock_27mhz, button_enter, beat_in_switch);
   debounce clean_which_beat(0, clock_27mhz, switch[2],which_beat);
   debounce clean_artificial(0, clock_27mhz, switch[3],artificial);
   assign beat_in = which_beat ? beat_in_switch : beat_in_video;
   wire[63:0] debug_data_write;
   wire[63:0] debug_data_read;
   wire[63:0] debug_data;
// assign led = 8'b11111111; currently assigned stuff in logic analyzer output,
don't uncomment wihtout changing that

   wire [639:0] dots;
   wire [2:0] read_state;

   assign debug_data = writemode ? debug_data_write : debug_data_read;
   display_16hex debug(reset, clock_27mhz, debug_data, disp_blank, disp_clock,
disp_rs, disp_ce_b,disp_reset_b, disp_data_out);
// display disp1      (reset, clock_27mhz, disp_blank, disp_clock, disp_rs,
disp_ce_b, disp_reset_b, disp_data_out, dots);

////////////////////////////////////////////////////
///USB//////////////////////////////////////////////
////////////////////////////////////////////////////
   wire[7:0] usb_byte;
   wire usb_has_new, usb_must_hold;
   wire[7:0] usb_data;
   wire usb_rxf, usb_rd, usb_reset;
   assign usb_data = user3[7:0];
   assign usb_rxf  = user3[9];
   assign user3[10]= ~usb_reset; //USB uses opposite reset from the rest of our
stuff
   assign user3[8] = usb_rd;
   assign usb_reset = reset;
   wire[3:0] usb_state;
   usb_input
usb245m(clock_27mhz,reset,usb_data,usb_rd,usb_rxf,usb_byte,usb_has_new,usb_must_hol
d,usb_state);
   //
usb_input(clk        ,reset,data    ,rd    ,rxf    ,out     ,newout     ,hold
);

/////////////////////////////////////////////////////
///Flash Memory Management///////////////////////////
/////////////////////////////////////////////////////
   wire mem_busy, mem_writemode;
   wire[22:0] raddr;          //no write address, it's a ROM so always increments
   wire[15:0] wdata, rdata;
   wire       write, read;
   wire[11:0] fsm_state;
   wire[3:0] accum_state;

   flash_manager
flash_man(clock_27mhz,reset,dots,writemode,wdata,write  ,raddr,rdata ,
read,mem_busy, flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
flash_reset_b, flash_sts, flash_byte_b, fsm_state);
   //
(clock      ,reset,dots,writemode,wdata,dowrite,raddr,frdata,doread,busy    ,
flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b,
flash_sts, flash_byte_b);

   data_accum
accumulator(clock_27mhz,reset,usb_byte,usb_has_new,usb_must_hold,debug_data_write,m
em_busy,write    ,wdata   ,accum_state);
   //
data_accum(clk        ,reset,data_in ,new_data   ,hold         ,hexdebug        ,me
m_busy,mem_write,mem_data);

/////////////////////////////////////////////////////
///Video SCHTUFF/////////////////////////////////////
/////////////////////////////////////////////////////
   wire[3:0] volume;// = 4'b1111;
   //wire video_beat_in;
   assign volume = user4[4:1];
   assign beat_in_video = user4[0];

/////////////////////////////////////////////////////
///Score Reading / Orchestra Playing/////////////////
/////////////////////////////////////////////////////
   wire new_continue,new_busy;
   wire[47:0] new_data;
   wire tick;
   wire[216:0] ptch_out;   //31 7-bit chunks
   wire[6:0] ptch [30:0];  //puts the bitch into chunks
   wire[30:0] enbl;        //31 1-bit chunks (already in chunks, thank god)
   wire[11:0] current_beat;

   tempo_synth temp_interp(clock_27mhz,reset,beat_in,tick,artificial);
   //          tempo_synth(clock      ,reset,beat_in,tick,artificial);

   score_man
score(clock_27mhz,writemode,new_continue,new_busy,new_data,tick   ,ptch_out,enbl
,current_beat,debug_data_read);
   //
(clock      ,reset    ,new_continue,new_busy,new_data,tick_in,ptch_out,enbl_out);

   //put our pitch array back into pitch form
   assign ptch[0] = ptch_out[  6:  0]; assign ptch[1] = ptch_out[ 13:  7]; assign
ptch[2] = ptch_out[ 20: 14];  assign ptch[3] = ptch_out[ 27: 21]; assign ptch[4] =
ptch_out[ 34: 28];   assign ptch[5] = ptch_out[ 41: 35]; assign ptch[6] = ptch_out[
48: 42]; assign ptch[7] = ptch_out[ 55: 49]; assign ptch[8] = ptch_out[ 62: 56];
   assign ptch[9] = ptch_out[ 69: 63]; assign ptch[10]= ptch_out[ 76: 70]; assign
ptch[11]= ptch_out[ 83: 77];  assign ptch[12]= ptch_out[ 90: 84]; assign ptch[13]=
ptch_out[ 97: 91];   assign ptch[14]= ptch_out[104: 98]; assign ptch[15]=
ptch_out[111:105];   assign ptch[16]= ptch_out[118:112]; assign ptch[17]=
ptch_out[125:119];   assign ptch[18]= ptch_out[132:126]; assign ptch[19]=
ptch_out[139:133];   assign ptch[20]= ptch_out[146:140]; assign ptch[21]=
ptch_out[153:147];   assign ptch[22]= ptch_out[160:154]; assign ptch[23]=
ptch_out[167:161];   assign ptch[24]= ptch_out[174:168]; assign ptch[25]=
ptch_out[181:175];   assign ptch[26]= ptch_out[188:182]; assign ptch[27]=
ptch_out[195:189];   assign ptch[28]= ptch_out[202:196]; assign ptch[29]=
ptch_out[209:203];   assign ptch[30]= ptch_out[216:210];

   mem_reader
reader(clock_27mhz,writemode,raddr,rdata,read,mem_busy,read_state,new_busy,new_cont
inue,new_data);   //
mem_reader(clock      ,reset,addr ,word ,read,mem_busy,hexdebug       ,state     ,b
usy    ,continue    ,data    );
   //
mem_reader(clock      ,reset    ,addr ,word ,read,mem_busy,state     ,busy    ,cont
inue    ,data    );

   orchestra
synth(clock_27mhz,writemode,volume               ,audio_reset_b,ac97_sdata_out,ac97
_sdata_in,ac97_synch,ac97_bit_clock,enbl[0]    ,enbl[1]    ,enbl[2]  ,enbl[3]  ,enb
l[4] ,ptch[0]      ,ptch[1]      ,ptch[2]    ,ptch[3]    ,ptch[4]   ,enbl[5]     ,e
nbl[6]     ,enbl[7] ,enbl[8]     ,enbl[9] ,ptch[5]       ,ptch[6]       ,ptch[7]
,ptch[8]       ,ptch[9]   ,enbl[10] ,enbl[11]    ,enbl[12],enbl[13],enbl[14]  ,ptch
[10]   ,ptch[11]      ,ptch[12]  ,ptch[13] ,ptch[14]    ,enbl[15]   ,enbl[16] ,enbl
[17] ,enbl[18] ,enbl[19]    ,ptch[15]     );
   //
orchestra(clock_27mhz,reset    ,volume_from_conductor,audio_reset_b,ac97_sdata_out,
ac97_sdata_in,ac97_synch,ac97_bit_clock,violin1_enb,violin2_enb,viola_enb,cello_enb
,bass_enb,violin1_pitch,violin2_pitch,viola_pitch,cello_pitch,bass_pitch,trumpet1_e
nb,trumpet2_enb,horn_enb,trombone_enb,tuba_enb,trumpet1_pitch,trumpet2_pitch,horn_p
itch,trombone_pitch,tuba_pitch,flute_enb,clarinet_enb,oboe_enb,sax_enb ,basoon_enb,
flute_pitch,clarinet_pitch,oboe_pitch,sax_pitch,basoon_pitch,timpani_enb,snare_enb,
hihat_enb,crash_enb,bassdrum_enb,timpani_pitch);

/////////////////////////////////////////////////////
///Logic Analyzer Output/////////////////////////////
/////////////////////////////////////////////////////
```

```
   assign led = {~current_beat[3:0],~enbl[3:0]};

//Debug States... //        A1  7 6           5              4          3     2
107        A06  5       4      3:0
   assign analyzer1_data =
{reset,usb_has_new,usb_must_hold,writemode,write,mem_busy,read_state,read,usb_rxf,u
sb_rd,usb_state};
   assign analyzer2_data = {raddr[3:0],rdata[3:0],wdata[7:0]};
   assign analyzer3_data = {fsm_state,accum_state};
   assign analyzer4_data = {fsm_state,accum_state};
// assign analyzer4_data = {tempo,6'd0};
// assign analyzer4_data = {usb_byte,usb_data};
// assign analyzer4_data =
{debug_data_read[51:48],debug_data_read[35:32],debug_data_read[19:16],4'b0000};

//Clocks...
   assign analyzer1_clock = clock_27mhz;
   assign analyzer3_clock = clock_27mhz;
   assign analyzer2_clock = clock_27mhz;
   assign analyzer4_clock = clock_27mhz;

endmodule
```

Appendix A: Score Data Transfer, Storage, and Playback: usb_input.v

```
//reads data and puts it on output pins
module usb_input(clk,reset,data,rd,rxf,out,newout,hold,state);
   input clk, reset;
   input [7:0] data;
   input rxf;
   output rd;
   reg rd;

   output[7:0] out;  //output data
   reg[7:0] out;
   output newout;    //lets outside modules know that there's new data
   reg newout;
   input hold;       //as long as hold is high, the module will not accept new data

   output state;
   reg[3:0] state;

   parameter RESET      = 0;     //state data
   parameter WAIT       = 1;
   parameter WAIT2         = 2;
   parameter WAIT3         = 3;
   parameter DATA_COMING   = 4;
   parameter DATA_COMING_2 = 5;
   parameter DATA_COMING_3 = 6;
   parameter DATA_COMING_4 = 7;
   parameter DATA_COMING_5 = 8;
   parameter DATA_HERE     = 9;
   parameter DATA_LEAVING  =10;
   parameter DATA_LEAVING_2=11;
   parameter DATA_LEAVING_3=12;

   initial
      state <= WAIT;
   always @ (posedge clk)
      if(reset)
         begin
            newout <= 0;
            rd <= 1;       //we can't read data
            state <= WAIT;
         end
      else
         if(~hold)
            begin
               newout <= 0;
               case(state)
               WAIT:
                  if(~rxf)    //if rxf is low and nobody's asking us to wait then
there is data waiting for us
                     begin
                        rd <= 1;              //so ask for it
                        state <= WAIT2;   //and start waiting for it
                     end

               WAIT2:
                  if(~rxf)    //double check
                     begin
                        rd <= 1;
                        state <= WAIT3;
                     end
                  else
                     state <= WAIT;

               WAIT3:
                  if(~rxf)    //and triple check (should only need one, but oh
well...)
                     begin
                        rd <= 0;
                        state <= DATA_COMING;
                     end
                  else
                     state <= WAIT;

               DATA_COMING:      //once rd goes low we gotta wait a bit for the
data to stabilize
                  state <= DATA_COMING_2;

               DATA_COMING_2:
                  state <= DATA_COMING_3;

               DATA_COMING_3:
                  state <= DATA_HERE;

               DATA_HERE:
                  begin
                     out <= data;   //the data is valid by now so read it
                     state <= DATA_LEAVING;
                     newout <= 1;   //let folks know we've got new data
                  end
               DATA_LEAVING:        //wait a cycle to clear the data to make sure
we latch onto it correctly
                  begin
                     rd <= 1;
                     state <= DATA_LEAVING_2;
                     newout <= 0;   //let folks know the data's a clock cycle old
now
                  end

               DATA_LEAVING_2:      //wait another cycle to make sure that the RD
to RD pre-charge time is met
                  state <= DATA_LEAVING_3;

               DATA_LEAVING_3:      //wait another cycle to make sure that the RD
to RD pre-charge time is met
                  state <= WAIT;

               default:
                  state <= WAIT;
            endcase
         end
endmodule
```

Appendix A: Score Data Transfer, Storage, and Playback: data_accum.v

```
//accumulates data into 2-byte words and writes them to flash
module
data_accum(clk,reset,data_in,new_data,hold,hexdebug,mem_busy,mem_write,mem_data,sta
te);
   input clk,reset;
   input[7:0] data_in;
   input new_data;
   output hold;
   reg hold;
   output[63:0] hexdebug;
   reg[63:0] hexdebug;
   reg[47:0] data;
   output state;
   reg[3:0] state;

   input mem_busy;
   output mem_write;

   output[15:0] mem_data;
   reg       mem_write;
   reg[15:0] mem_data;

   parameter PREP_FOR_READ = 0;
   parameter DATA1 = 1;
   parameter DATA2 = 2;
   parameter WRITE1= 3;
   parameter DATA3 = 4;
   parameter DATA4 = 5;
   parameter WRITE2= 6;
   parameter DATA5 = 7;
   parameter DATA6 = 8;
   parameter WRITE3= 9;
   initial
      state <= PREP_FOR_READ;

   always @ (posedge clk)
      if(reset)
         begin
            mem_write<= 0;
            hold <= 1;           //if we're in reset then make sure they don't
leave without us
            state <= PREP_FOR_READ; //move to the first data state
         end
      else
      case(state)
         PREP_FOR_READ://0
            begin
               hold <= 0;
               state <= DATA1;
            end

         DATA1://1
            begin
               hold <= 0;
               mem_write <= 0;
               if(new_data)            //if there's new data to be had
                  begin
                     data[47:40] <= data_in; //grab it
                     state <= DATA2;
                  end
            end

         DATA2://2
            if(new_data)            //if there's new data to be had
               begin
                  data[39:32] <= data_in; //grab it
                  hold <= 1;        //then we wait
                  state <= WRITE1;
               end

         WRITE1://3        //so now we've read two bytes (one address)
            if(!mem_busy)
               begin       //if the memory is ready
                  mem_write <= 1;            //we ask it to write
                  mem_data <= data[47:32];   //and we give it the data
                  state <= DATA3;         //and we can start getting the new data
from the USB
               end

         DATA3://4
            begin
               hold <= 0;
               mem_write <= 0;
               if(new_data)            //if there's new data to be had
                  begin
                     data[31:24] <= data_in; //grab it
                     state <= DATA4;
                  end
            end
         DATA4://5
            if(new_data)             //if there's new data to be had
               begin
                  data[23:16] <= data_in; //grab it
                  state <= WRITE2;
                  hold <= 1;        //then we wait
               end

         WRITE2://6        //so now we've read two bytes (one address)
            if(!mem_busy)        //if the memory isn't ready to write the new stuff
               begin       //but if it is ready
                  mem_write <= 1;            //we ask it to write
                  mem_data <= data[31:16];   //and we give it the data
                  state <= DATA5;         //and we can start getting the new data
from the USB
               end

         DATA5://7
            begin
               hold <= 0;
               mem_write <= 0;
               if(new_data)             //if there's new data to be had
                  begin
                     data[15:8] <= data_in;  //grab it
                     state <= DATA6;
                  end
            end

         DATA6://8
            if(new_data)             //if there's new data to be had
               begin
                  data[7:0] <= data_in;   //grab it
                  hold <= 1;
                  state <= WRITE3;
               end

         WRITE3://9
            if(!mem_busy)        //if the memory is ready to write the new stuff
               begin
                  mem_write <= 1;            //we ask it to write
                  mem_data <= data[15:0]; //and we give it the data
                  state <= DATA1;         //and we can start getting the new data
from the USB
                  hexdebug <= {8'b00000000,8'b00000000,4'b0000,data[47:36],
1'b0,data[35:29],1'b0,data[28:22],1'b0,data[21:15],7'b0000000,data[14]};
                  //hexdebug <= {data[47:0],16'b0000000000000000};
               end
               //beat = data[47:36];
               //tick = data[35:29];
               //inst = data[28:22];
               //ptch = data[21:15];
               //enbl = data[14];
      endcase
endmodule
```

Appendix A: Score Data Transfer, Storage, and Playback: flash_manager.v

```
//manages all the stuff needed to read and write to the flash ROM
module flash_manager(clock, reset, dots, writemode, wdata, dowrite, raddr, frdata,
doread, busy, flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
flash_reset_b, flash_sts, flash_byte_b, fsmstate);

   input reset, clock;

   output [639:0] dots;

   input writemode;           //if true then we're in write mode, else we're in
read mode
   input [15:0] wdata;
   input dowrite;
   input [22:0] raddr;     //inputs for reading data
   output[15:0] frdata;
   reg[15:0]    rdata;
   input doread;

   output busy;            //and an output to tell folks we're still working on the
last thing
   reg busy;

   inout [15:0] flash_data;                    //direct passthrough to the flash
stuff
    output [23:0] flash_address;
    output flash_ce_b, flash_oe_b, flash_we_b;
    output flash_reset_b, flash_byte_b;
    input  flash_sts;

   wire flash_busy;     //except these, which are internal to the interface
   wire[15:0] fwdata;
   wire[15:0] frdata;
   wire[22:0] address;
   wire [1:0] op;

   reg [1:0] mode;
   wire fsm_busy;

   reg[2:0] state;               //210

   output[11:0] fsmstate;
   wire [7:0] fsmstateinv;
// assign fsmstate = {~busy,~flash_sts,~flash_busy,~fsmstateinv[4:0]};
   assign fsmstate = {state,flash_busy,fsm_busy,fsmstateinv[4:0],mode}; //for
debugging only

                              //this guy takes care of /some/ of flash's tantrums
   flash_int flash(reset, clock, op, address, fwdata, frdata, flash_busy,
flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b,
flash_sts, flash_byte_b);
                              //and this guy takes care of the rest of its tantrums
   test_fsm  fsm  (reset, clock, op, address, fwdata, frdata, flash_busy, dots,
mode, fsm_busy, wdata, raddr, fsmstateinv);
//       test_fsm(reset, clock, op, address, fwdata, frdata, flash_busy, dots,
mode, fsm_busy, wdata, raddr);
   parameter MODE_IDLE  = 0;
   parameter MODE_INIT  = 1;
   parameter MODE_WRITE = 2;
   parameter MODE_READ  = 3;

   parameter HOME       = 3'd0;
   parameter MEM_INIT   = 3'd1;
   parameter MEM_WAIT   = 3'd2;
   parameter WRITE_READY= 3'd3;
   parameter WRITE_WAIT = 3'd4;
   parameter READ_READY = 3'd5;
   parameter READ_WAIT  = 3'd6;

   always @ (posedge clock)
      if(reset)
         begin
            busy <= 1;
            state <= HOME;
            mode <= MODE_IDLE;
         end
      else begin
         case(state)
            HOME://0          //we always start here
               if(!fsm_busy)
                  begin
                     busy <= 0;
                     if(writemode)
                        begin
                           busy <= 1;
                           state <= MEM_INIT;
                        end
                     else
                        begin
                           busy <= 1;
                           state <= READ_READY;
                        end
                  end
               else
                  mode <= MODE_IDLE;

            MEM_INIT://1                    //begin wiping the memory
               begin
                  busy <= 1;
                  mode <= MODE_INIT;
                  if(fsm_busy)              //to give the fsm a chance to raise
its busy signal
                     state <= MEM_WAIT;
               end

            MEM_WAIT://2                 //finished wiping
               if(!fsm_busy)
                  begin
                     busy <= 0;
                     state<= WRITE_READY;
                  end
               else
                  mode <= MODE_IDLE;
            WRITE_READY://3               //waiting for data to write to flash
               if(dowrite)
                  begin
                     busy <= 1;
                     mode <= MODE_WRITE;
                  end
               else if(busy)
                  state <= WRITE_WAIT;
               else if(!writemode)
                  state <= READ_READY;

            WRITE_WAIT://4          //waiting for flash to finish writing
               if(!fsm_busy)
                  begin
                     busy <= 0;
                     state <= WRITE_READY;
                  end
               else
                  mode <= MODE_IDLE;

            READ_READY://5          //ready to read data
               if(doread)
                  begin
                     busy <= 1;
                     mode <= MODE_READ;
                     if(busy)       //lets the fsm raise its busy level
                        state <= READ_WAIT;
                  end
               else
                  busy <= 0;

            READ_WAIT://6        //waiting for flash to give the data up
               if(!fsm_busy)
                  begin
                     busy <= 0;
                     state <= READ_READY;
                  end
               else
                  mode <= MODE_IDLE;

            default: begin    //should never happen...
               state <= 3'd7;
            end
         endcase
   end
endmodule
```

Appendix A: Score Data Transfer, Storage, and Playback: test_fsm.v

```
`define STATUS_RESET             4'h0
`define STATUS_READ_ID           4'h1
`define STATUS_CLEAR_LOCKS       4'h2
`define STATUS_ERASING           4'h3
`define STATUS_WRITING           4'h4
`define STATUS_READING           4'h5
`define STATUS_SUCCESS           4'h6
`define STATUS_BAD_MANUFACTURER  4'h7
`define STATUS_BAD_SIZE          4'h8
```

```
`define STATUS_LOCK_BIT_ERROR    4'h9
`define STATUS_ERASE_BLOCK_ERROR 4'hA
`define STATUS_WRITE_ERROR       4'hB
`define STATUS_READ_WRONG_DATA   4'hC

`define NUM_BLOCKS 128
`define BLOCK_SIZE 64*1024
`define LAST_BLOCK_ADDRESS ((`NUM_BLOCKS-1)*`BLOCK_SIZE)
`define LAST_ADDRESS (`NUM_BLOCKS*`BLOCK_SIZE-1)

`define FLASHOP_IDLE  2'b00
`define FLASHOP_READ  2'b01
`define FLASHOP_WRITE 2'b10

module test_fsm (reset, clock, fop, faddress, fwdata, frdata, fbusy, dots, mode,
busy, datain, addrin, state);
   input reset, clock;
   output [1:0] fop;
   output [22:0] faddress;
   output [15:0] fwdata;
   input [15:0]  frdata;
   input fbusy;
   output [639:0] dots;
   input [1:0] mode;
   output busy;
   input [15:0] datain;
   input [22:0] addrin;
   output state;

   reg [1:0] fop;
   reg [22:0] faddress;
   reg [15:0] fwdata;
   reg [639:0] dots;
   reg busy;
   reg [15:0] data_to_store;

   ////////////////////////////////////////////////////////////////////////////
   //
   // State Machine
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [7:0] state;
   reg [3:0] status;

   parameter MODE_IDLE  = 0;
   parameter MODE_INIT  = 1;
   parameter MODE_WRITE = 2;
   parameter MODE_READ  = 3;

   parameter MAX_ADDRESS = 23'h030000;

   parameter HOME = 8'h12;


   always @(posedge clock)
     if (reset)
       begin
         state <= HOME;
         status <= `STATUS_RESET;
         faddress <= 0;
         fop <= `FLASHOP_IDLE;
         busy <= 1;
       end
     else if (!fbusy && (fop == `FLASHOP_IDLE))
       case (state)

    HOME://12
      case(mode)
         MODE_INIT: begin
            state <= 8'h00;
            busy <= 1;
         end

         MODE_WRITE: begin
            state <= 8'h0C;
            busy <= 1;
         end

         MODE_READ: begin
            busy <= 1;
            if(status == `STATUS_READING)
               state <= 8'h11;
            else
               state <= 8'h10;
         end

         default: begin
            state <= HOME;
            busy <= 0;
         end
      endcase

    //////////////////////////////////////////////////////////////////////
    // Wipe It
    //////////////////////////////////////////////////////////////////////
    8'h00:
      begin
         // Issue "read id codes" command
         status <= `STATUS_READ_ID;
         faddress <= 0;
         fwdata <= 16'h0090;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end

    8'h01:
      begin
         // Read manufacturer code
         faddress <= 0;
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
```

```
8'h02:
  if (frdata != 16'h0089) // 16'h0089 = Intel
    status <= `STATUS_BAD_MANUFACTURER;
  else
    begin
  // Read the device size code
  faddress <= 1;
  fop <= `FLASHOP_READ;
  state <= state+1;
    end

8'h03:
  if (frdata != 16'h0018) // 16'h0018 = 128Mbit
    status <= `STATUS_BAD_SIZE;
  else
    begin
  faddress <= 0;
  fwdata <= 16'hFF;
  fop <= `FLASHOP_WRITE;
  state <= state+1;
    end

8'h04:
  begin
     // Issue "clear lock bits" command
     status <= `STATUS_CLEAR_LOCKS;
     faddress <= 0;
     fwdata <= 16'h60;
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end

8'h05:
  begin
     // Issue "confirm clear lock bits" command
     faddress <= 0;
     fwdata <= 16'hD0;
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end

8'h06:
  begin
     // Read status
     faddress <= 0;
     fop <= `FLASHOP_READ;
     state <= state+1;
  end

8'h07:
  if (frdata[7] == 1) // Done clearing lock bits
    if (frdata[6:1] == 0) // No errors
      begin
    faddress <= 0;
    fop <= `FLASHOP_IDLE;
    state <= state+1;
      end
    else
      status <= `STATUS_LOCK_BIT_ERROR;
  else // Still busy, reread status register
    begin
  faddress <= 0;
  fop <= `FLASHOP_READ;
    end

//////////////////////////////////////////////////////////////////////
// Block Erase Sequence
//////////////////////////////////////////////////////////////////////
8'h08:
  begin
     status <= `STATUS_ERASING;
     fwdata <= 16'h20; // Issue "erase block" command
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end

8'h09:
  begin
     fwdata <= 16'hD0; // Issue "confirm erase" command
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end
8'h0A:
  begin
     fop <= `FLASHOP_READ;
     state <= state+1;
  end
8'h0B:
  if (frdata[7] == 1) // Done erasing block
    if (frdata[6:1] == 0) // No errors
      if (faddress != MAX_ADDRESS) // `LAST_BLOCK_ADDRESS)
   begin
      faddress <= faddress+`BLOCK_SIZE;
      fop <= `FLASHOP_IDLE;
      state <= state-3;
   end
      else
   begin
      faddress <= 0;
      fop <= `FLASHOP_IDLE;
      state <= HOME;      //done erasing, go home
   end
    else // Erase error detected
      status <= `STATUS_ERASE_BLOCK_ERROR;
  else // Still busy
    fop <= `FLASHOP_READ;

//////////////////////////////////////////////////////////////////////
// Write Mode
//////////////////////////////////////////////////////////////////////
8'h0C:
  begin
     data_to_store <= datain;
     status <= `STATUS_WRITING;
```

```
      fwdata <= 16'h40; // Issue "setup write" command
      fop <= `FLASHOP_WRITE;
      state <= state+1;
   end

  8'h0D:
   begin
      fwdata <= data_to_store; // Finish write
      fop <= `FLASHOP_WRITE;
      state <= state+1;
   end
  8'h0E:
   begin
      // Read status register
      fop <= `FLASHOP_READ;
      state <= state+1;
   end
  8'h0F:
   if (frdata[7] == 1) // Done writing
     if (frdata[6:1] == 0) // No errors
       if (faddress != 23'h7FFFFF) // `LAST_ADDRESS)
         begin
            faddress <= faddress+1;
            fop <= `FLASHOP_IDLE;
            state <= HOME;
         end
       else
         status <= `STATUS_WRITE_ERROR;
     else // Write error detected
       status <= `STATUS_WRITE_ERROR;
   else // Still busy
     fop <= `FLASHOP_READ;

  ///////////////////////////////////////////////////////////////////////
  // Read Mode INIT
  ///////////////////////////////////////////////////////////////////////
  8'h10:
   begin
      status <= `STATUS_READING;
      fwdata <= 16'hFF; // Issue "read array" command
      fop <= `FLASHOP_WRITE;
      faddress <= 0;
      state <= state+1;
   end

  ///////////////////////////////////////////////////////////////////////
  // Read Mode
  ///////////////////////////////////////////////////////////////////////
  8'h11:
   begin
      faddress <= addrin;
      fop <= `FLASHOP_READ;
      state <= HOME;
   end

 default:
   begin
```

```
      status <= `STATUS_BAD_MANUFACTURER;
      faddress <= 0;
      state <= HOME;
   end

endcase
  else
    fop <= `FLASHOP_IDLE;

function [39:0] nib2char;
   input [3:0] nib;
   begin
 case (nib)
   4'h0: nib2char = 40'b00111110_01010001_01001001_01000101_00111110;
   4'h1: nib2char = 40'b00000000_01000010_01111111_01000000_00000000;
   4'h2: nib2char = 40'b01100010_01010001_01001001_01001001_01000110;
   4'h3: nib2char = 40'b00100010_01000001_01001001_01001001_00110110;
   4'h4: nib2char = 40'b00011000_00010100_00010010_01111111_00010000;
   4'h5: nib2char = 40'b00100111_01000101_01000101_01000101_00111001;
   4'h6: nib2char = 40'b00111100_01001010_01001001_01001001_00110000;
   4'h7: nib2char = 40'b00000001_01110001_00001001_00000101_00000011;
   4'h8: nib2char = 40'b00110110_01001001_01001001_01001001_00110110;
   4'h9: nib2char = 40'b00000110_01001001_01001001_00101001_00011110;
   4'hA: nib2char = 40'b01111110_00001001_00001001_00001001_01111110;
   4'hB: nib2char = 40'b01111111_01001001_01001001_01001001_00110110;
   4'hC: nib2char = 40'b00111110_01000001_01000001_01000001_00100010;
   4'hD: nib2char = 40'b01111111_01000001_01000001_01000001_00111110;
   4'hE: nib2char = 40'b01111111_01001001_01001001_01001001_01000001;
   4'hF: nib2char = 40'b01111111_00001001_00001001_00001001_00000001;
 endcase
   end
endfunction

wire [159:0] data_dots;
assign data_dots = {nib2char(frdata[15:12]), nib2char(frdata[11:8]),
          nib2char(frdata[7:4]), nib2char(frdata[3:0])};

wire [239:0] address_dots;
assign address_dots = {nib2char({ 1'b0, faddress[22:20]}),
        nib2char(faddress[19:16]),
        nib2char(faddress[15:12]),
        nib2char(faddress[11:8]),
        nib2char(faddress[7:4]),
        nib2char(faddress[3:0])};

always @(status or address_dots or data_dots)
  case (status)
    `STATUS_RESET:
 dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
     40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b00100110_01001001_01001001_01001001_00110010, // S
     40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b00000001_00000001_01111111_00000001_00000001, // T
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
```

```
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000};// -
       `STATUS_READ_ID:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_CLEAR_LOCKS:
    dots <= {40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b00111110_01000001_01000001_01000001_00111110, // O
        40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_00001000_00010100_00100010_01000001, // K
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_ERASING:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_WRITING:
    dots <= {40'b01111111_00100000_00011000_00100000_01111111, // W
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b00000001_00000001_01111111_00000001_00000001, // T
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_READING:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
```

```
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_SUCCESS:
    dots <= {40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00001001_00001001_00001001_00000110, // P
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00000000_00000000_00000000_00000000_00000000};//
       `STATUS_BAD_MANUFACTURER:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00000010_00001100_00000010_01111111, // M
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b01111111_00001001_00001001_00001001_00000001, // U
        40'b01111111_00001001_00001001_00001001_00000001, // F
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        data_dots};
       `STATUS_BAD_SIZE:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01100001_01010001_01001001_01000101_01000011, // Z
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_LOCK_BIT_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
```

```
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b00111110_01000001_01000001_01000001_00111110, // O
        40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_00001000_00010100_00100010_01000001, // K
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_ERASE_BLOCK_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_WRITE_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00100000_00011000_00100000_01111111, // W
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b00000001_00000001_01111111_00000001_00000001, // T
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_READ_WRONG_DATA:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000,
        address_dots,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
    default:
    dots <= {16{40'b01010101_00101010_01010101_00101010_01010101}};
     endcase
endmodule
```

Appendix A: Score Data Transfer, Storage, and Playback: flash_int.v

```
//flash interface
module flash_int(reset, clock, op, address, wdata, rdata, busy, flash_data,
       flash_address, flash_ce_b, flash_oe_b, flash_we_b,
       flash_reset_b, flash_sts, flash_byte_b);

   parameter access_cycles = 5;
   parameter reset_assert_cycles = 1000;
   parameter reset_recovery_cycles = 30;

   input reset, clock; // Reset and clock for the flash interface
   input [1:0] op; // Flash operation select (read, write, idle)
   input [22:0] address;
   input [15:0] wdata;
   output [15:0] rdata;
   output busy;
   inout [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b;
   output flash_reset_b, flash_byte_b;
   input  flash_sts;

   reg [1:0] lop;
   reg [15:0] rdata;
   reg busy;
   reg [15:0] flash_wdata;
   reg flash_ddata;
   reg [23:0] flash_address;
   reg flash_oe_b, flash_we_b, flash_reset_b;

   assign flash_ce_b = flash_oe_b && flash_we_b;
   assign flash_byte_b = 1; // 1 = 16-bit mode (A0 ignored)

   assign flash_data = flash_ddata ? flash_wdata : 16'hZ;

   initial
     flash_reset_b <= 1'b1;

   reg [9:0] state;

   always @(posedge clock)
     if (reset)
       begin
         state <= 0;
         flash_reset_b <= 0;
         flash_we_b <= 1;
         flash_oe_b <= 1;
         flash_ddata <= 0;
         busy <= 1;
       end
     else if (flash_reset_b == 0)
       if (state == reset_assert_cycles)
         begin
            flash_reset_b <= 1;
            state <= 1023-reset_recovery_cycles;
         end
       else
         state <= state+1;
    else if ((state == 0) && !busy)
       // The flash chip and this state machine are both idle. Latch the user's
       // address and write data inputs. Deassert OE and WE, and stop driving
       // the data buss ourselves. If a flash operation (read or write) is
       // requested, move to the next state.
       begin
         flash_address <= {address, 1'b0};
         flash_we_b <= 1;
         flash_oe_b <= 1;
         flash_ddata <= 0;
         flash_wdata <= wdata;
         lop <= op;
         if (op != `FLASHOP_IDLE)
            begin
               busy <= 1;
               state <= state+1;
            end
         else
            busy <= 0;
         end
      else if ((state==0) && flash_sts)
         busy <= 0;
      else if (state == 1)
       // The first stage of a flash operation. The address bus is already set,
       // so, if this is a read, we assert OE. For a write, we start driving
       // the user's data onto the flash databus (the value was latched in the
       // previous state.
         begin
            if (lop == `FLASHOP_WRITE)
               flash_ddata <= 1;
            else if (lop == `FLASHOP_READ)
               flash_oe_b <= 0;
            state <= state+1;
       end
     else if (state == 2)
       // The second stage of a flash operation. Nothing to do for a read. For
       // a write, we assert WE.
       begin
         if (lop == `FLASHOP_WRITE)
            flash_we_b <= 0;
         state <= state+1;
       end
     else if (state == access_cycles+1)
       // The third stage of a flash operation. For a read, we latch the data
       // from the flash chip. For a write, we deassert WE.
       begin
         if (lop == `FLASHOP_WRITE)
            flash_we_b <= 1;
         if (lop == `FLASHOP_READ)
            rdata <= flash_data;
         state <= 0;
       end
     else
       begin
         if (!flash_sts)
            busy <= 1;
         state <= state+1;
       end
```

```
endmodule
```

Appendix A: Score Data Transfer, Storage, and Playback: score_man.v

```
module
score_man(clock,reset,new_continue,new_busy,new_data,tick_in,ptch_out,enbl_out,curr
ent_beat,hexdebug);
   input clock, reset;

   output new_continue;
   reg new_continue;
   input new_busy;
   input[47:0] new_data;

   input tick_in;

   output[216:0] ptch_out;
   output[30:0] enbl_out;
   reg[6:0] ptch_all[30:0];   //should be 127:0, but we're only actually going to
have ~30 built by now
   reg      enbl_all[30:0];
   assign ptch_out =
{ptch_all[30],ptch_all[29],ptch_all[28],ptch_all[27],ptch_all[26],ptch_all[25],ptch
_all[24],ptch_all[23],ptch_all[22],ptch_all[21],ptch_all[20],ptch_all[19],ptch_all[
18],ptch_all[17],ptch_all[16],ptch_all[15],ptch_all[14],ptch_all[13],ptch_all[12],p
tch_all[11],ptch_all[10],ptch_all[9],ptch_all[8],ptch_all[7],ptch_all[6],ptch_all[5
],ptch_all[4],ptch_all[3],ptch_all[2],ptch_all[1],ptch_all[0]};
   assign enbl_out =
{enbl_all[30],enbl_all[29],enbl_all[28],enbl_all[27],enbl_all[26],enbl_all[25],enbl
_all[24],enbl_all[23],enbl_all[22],enbl_all[21],enbl_all[20],enbl_all[19],enbl_all[
18],enbl_all[17],enbl_all[16],enbl_all[15],enbl_all[14],enbl_all[13],enbl_all[12],e
nbl_all[11],enbl_all[10],enbl_all[9],enbl_all[8],enbl_all[7],enbl_all[6],enbl_all[5
],enbl_all[4],enbl_all[3],enbl_all[2],enbl_all[1],enbl_all[0]};

   reg[47:0] current_data;
   reg oldtick;
   output[11:0] current_beat;
   reg[11:0] current_beat;
   reg[6:0] current_tick;

   output [63:0] hexdebug;

   wire[11:0] beat;
   wire [6:0] tick;
   wire [6:0] inst;
   wire [6:0] ptch;
   wire       enbl;

   assign beat = current_data[47:36];
   assign tick = current_data[35:29];
   assign inst = current_data[28:22];
   assign ptch = current_data[21:15];
   assign enbl = current_data[14];  //bits 13-0 unused at the moment

   assign hexdebug = {current_beat[7:0],1'b0,current_tick,
4'b0000,current_data[47:36],1'b0,current_data[35:29],1'b0,current_data[28:22],
1'b0,current_data[21:15],7'b0000000,current_data[14]};
```

```
   reg state;
   parameter DO_NOTE   = 0;
   parameter LOAD_NOTE = 1;

   always @ (posedge clock)
      if(reset)
         begin
            state <= LOAD_NOTE;
            enbl_all[0] <= 0;           enbl_all[1] <= 0;           enbl_all[2] <= 0;
            enbl_all[3] <= 0;           enbl_all[4] <= 0;           enbl_all[5] <= 0;
            enbl_all[6] <= 0;           enbl_all[7] <= 0;           enbl_all[8] <= 0;
            enbl_all[9] <= 0;           enbl_all[10] <= 0;             enbl_all[11]
<= 0;          enbl_all[12] <= 0;            enbl_all[13] <= 0;
   enbl_all[14] <= 0;            enbl_all[15] <= 0;            enbl_all[16] <= 0;
            enbl_all[17] <= 0;            enbl_all[18] <= 0;
   enbl_all[19] <= 0;            enbl_all[20] <= 0;            enbl_all[21] <= 0;
            enbl_all[22] <= 0;            enbl_all[23] <= 0;
   enbl_all[24] <= 0;            enbl_all[25] <= 0;            enbl_all[26] <= 0;
            enbl_all[27] <= 0;            enbl_all[28] <= 0;
   enbl_all[29] <= 0;            enbl_all[30] <= 0;
            current_beat <= 0;
            current_tick <= 0;
            oldtick <= tick_in;
         end
      else
         begin
            //keeping track of time executes in parallel with playing notes
            if(tick_in!=oldtick)       //if the beat changed
               begin
                  oldtick <= tick_in;     //update our memory
                  current_tick <= current_tick + 1;   //increment tick
                  if(current_tick == 7'd127)       //and if necessary increment
beat as well
                     current_beat <= current_beat + 1;   //and increment the
current beat
               end

            case(state)
               DO_NOTE:
                  begin
                     new_continue <= 0;
                     if((beat <= current_beat) && (tick <= current_tick))  //if the
note should happen
                        begin
                           ptch_all[inst] <= ptch;    //make it happen
                           enbl_all[inst] <= enbl;
                           state <= LOAD_NOTE;        //and get the next note
                        end
                  end

               LOAD_NOTE:
                  if(~new_busy)     //if the memory's ready
                     begin
                        current_data <= new_data;  //send over the data
                        new_continue <= 1;         //tell the memory it's good to
go
```

```
                        state <= DO_NOTE;          //and move on with our life
                     end
            endcase
         end
endmodule
```

Appendix A: Score Data Transfer, Storage, and Playback: mem_reader.v

```
module mem_reader(clock,reset,addr,word,read,mem_busy,state,busy,continue,data);
   input clock, reset;
   output [22:0] addr;
   input [15:0] word;
   output read;
   input mem_busy;
   reg [22:0] addr;
   reg read;
   output[2:0] state;
   output busy;
   input continue;
   output[47:0] data;
   reg busy;
   reg[2:0] state;
   reg[47:0] data;

   parameter HOME       = 0;
   parameter PRE_READ1  = 1;
   parameter READ1      = 2;
   parameter PRE_READ2  = 3;
   parameter READ2      = 4;
   parameter PRE_READ3  = 5;
   parameter READ3      = 6;
   parameter WAIT       = 7;

   always @ (posedge clock)
      if(reset)
         begin
            addr <= 0;
            read <= 0;
            state <= HOME;
            busy <= 1;
         end
      else

      case(state)
         HOME://0
            if(!(mem_busy || reset))
               begin
                  state <= PRE_READ1;
                  read <= 1;
               end

         PRE_READ1://1
            state <= READ1;

         READ1://2
            begin
               read <= 0;
               if(!mem_busy)
                  begin
                     data[47:32] <= word;
                     addr <= addr + 1;
                     state <= PRE_READ2;
                     read <= 1;
                  end
            end

         PRE_READ2://3
            state <= READ2;

         READ2://4
            begin
               read <= 0;
               if(!mem_busy)
                  begin
                     data[31:16] <= word;
                     addr <= addr + 1;
                     state <= PRE_READ3;
                     read <= 1;
                  end
            end

         PRE_READ3://5
            state <= READ3;

         READ3://6
            begin
               read <= 0;
               if(!mem_busy)
                  begin
                     data[15:0] <= word;
                     addr <= addr + 1;
                     state <= WAIT;
                  end
            end

         WAIT://7
            begin
               busy <= 0;
               if(continue)          //if the scoreman took the note
                  begin
                     state <= PRE_READ1;  //go get the next one
                     read <= 1;
                     busy <= 1;        //and make sure the scoreman knows we're not
ready for him to take the new one
                  end
            end
      endcase
endmodule
```

Appendix A: Score Data Transfer, Storage, and Playback: tempo_synth.v

```
module tempo_synth(clock,reset,beat_in,tick,artificial);

   parameter MAX_BEAT_COUNT = 32'd4294967295;
   parameter MAX_TICK_PERIOD = 25'd33554431;
   input clock, reset;
   input beat_in,artificial;
   output tick;

   reg oldbeat;
   reg tick;

   reg [17:0] static_counter;
   reg [24:0] tick_period;
   reg [25:0] tick_counter;
   reg [31:0] beat_counter;

   always @ (posedge clock) begin

      if (artificial) begin
         if (static_counter == 18'd210937) begin
            static_counter <= 0;
            tick <= ~tick;
         end
         else static_counter <= static_counter + 1;
      end

      else begin
         if(beat_in == ~oldbeat) begin
            oldbeat <= beat_in;
            if(beat_counter > 9000000) //sets maximum tempo to 180bpm
               begin
                  tick_period <= (beat_counter >> 7);
                  beat_counter <= 0;
               end
         end
         else begin
            if (beat_counter == MAX_BEAT_COUNT)
               beat_counter <= MAX_BEAT_COUNT;
            else beat_counter <= beat_counter + 1;
         end


         if(tick_counter > tick_period) begin
            if (tick_period == MAX_TICK_PERIOD) begin
               tick_period <= MAX_TICK_PERIOD;
               tick_counter <= 0;
            end

            else begin
            tick_counter <= 0;
            tick <= ~tick;
            end
         end

         else tick_counter <= tick_counter + 1;
      end
   end
endmodule
```

Appendix A: Audio Synthesis: brass_manager.v

```
// This module is the skeleton of all oscillator modules.
// Depending on the instrument, the original frequency at
// which the sample was recorded changes, and therefore the
// period table used in this module would be different. Hence
// all of the oscillator modules (string_oscillator,
// brass_oscillator, etc) are versions of this module with
// different period table BRAMs.
//
// This module takes in a pitch, and calculates the address
// of the next sample that corresponds with this pitch.


module oscillator(clock_27mhz, ready, pitch, sample_table_size, envelope_done,
addr, oscillator_done);

   parameter READY_SIGNAL_PERIOD = 10'd563;

   // FSM Parameters
   parameter STAND_BY_FOR_READY = 2'd0;
   parameter STAND_BY_FOR_ENVELOPE_DONE = 2'd1;
   parameter CALCULATE_ADDRESS = 2'd2;

   input clock_27mhz;
   input ready;
   input [6:0] pitch;
   input [15:0] sample_table_size; // Size of the sample table BRAM
   input envelope_done;
   output [15:0] addr;
   output oscillator_done;

   reg [15:0] addr;
   reg oscillator_done;
   reg [31:0] remainder;
   reg [1:0] state;

   wire [16:0] period;

   // Period table BRAM - list of period values for different pitches
   period_table_60 period_table(pitch, clock_27mhz, period);

   initial addr <= 0;
   initial oscillator_done <= 0;
   initial remainder <= 32'd563;
   initial state <= STAND_BY_FOR_READY;


   always @ (posedge clock_27mhz) begin

   case (state)

      // Wait until the next ready pulse
      STAND_BY_FOR_READY:
         begin
            if (ready) begin
               oscillator_done <= 0;
               state <= STAND_BY_FOR_ENVELOPE_DONE;
            end
         end

      // Wait until the previous instrument's envelope is done
      // (brass and strings only. There will be no wait here in
      // the case of woodwinds, as the calculation of the audio
      // signals is simultaneous, not sequential).
      STAND_BY_FOR_ENVELOPE_DONE:
         begin
            if (envelope_done) state <= CALCULATE_ADDRESS;
         end

      // Given the pitch, calculate the next address of the sample
      // by looking at how many times the address must have
      // incremented between the last and the next ready pulse.
      CALCULATE_ADDRESS:
         begin
            if (remainder >= period) begin
               remainder <= remainder - period;
               if (addr == sample_table_size) addr <= 0;
               else addr <= addr + 1;
            end
            else begin
               remainder <= remainder + READY_SIGNAL_PERIOD;
               oscillator_done <= 1;
               state <= STAND_BY_FOR_READY;
            end
         end

      endcase
   end
endmodule
```

Appendix A: Audio Synthesis: dac.v

```
// A digital-to-analog converter module which acts as a
// wrapper for the AC97 module. The AC97 module is a
// modified version of AC97 from lab 4, and it now supports
// 16-bit signals instead of 8-bit signals.
//
// This module takes in the audio signals generated by the
// instrument managers, and converts them to appropriate
// signals to be sent to AC97.

module dac(clock_27mhz, reset, volume, audio_out_left,
            audio_out_right, ready,
            audio_reset_b, ac97_sdata_out, ac97_sdata_in,
            ac97_synch, ac97_bit_clock);

   input clock_27mhz;
   input reset;
   input [4:0] volume; // volume specified by the conductor
   input [15:0] audio_out_left; // left-channel audio from the mixer module
   input [15:0] audio_out_right;// right-channel audio from the mixer module
   output ready; // ready signal requesting a sample

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

   audio16bit audio_output(clock_27mhz, reset, volume,
                           audio_out_left, audio_out_right, ready,
                           audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                           ac97_synch, ac97_bit_clock);



endmodule



////////////////////////////////////////////////////////////////////////////////
//
// uni-directional stereo interface to AC97
//
////////////////////////////////////////////////////////////////////////////////

module audio16bit (clock_27mhz, reset, volume,
                  audio_out_left, audio_out_right, ready,
                  audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                  ac97_synch, ac97_bit_clock);

   input clock_27mhz;
   input reset;
   input [4:0] volume;
   //output [15:0] audio_in_data;
   input [15:0] audio_out_left;
   input [15:0] audio_out_right;
   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 [2:0] source;
   assign source = 0;      //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

   wire ac97_ready;
   ac97 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);

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

      //////////////////////////////////////////////////////////////////
      // STEREO SETTINGS
   //////////////////////////////////////////////////////////////////

   reg [15:0] left_data, right_data;

   always @ (posedge clock_27mhz) begin
     if (ready) begin
      left_data <= audio_out_left;
      right_data <= audio_out_right;
     end
   end

   //assign audio_in_data = left_in_data[19:12];
   assign left_out_data = {left_data, 4'b0000};
   assign right_out_data = {right_data, 4'b0000};

   // generate repeating sequence of read/writes to AC97 registers
   ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                     command_valid, volume, source);
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 [2:0] source;

   reg [23:0] command;
   reg command_valid;

   reg [3:0] state;

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

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

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

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

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

```
          command <= 24'h20_8000;
        default:
          command <= 24'h80_0000;
      endcase // case(state)
   end // always @ (posedge clock)
endmodule // ac97commands
```

Appendix A: Audio Synthesis: envelope.v

```
// This is an ADSR envelope (Attack, Decay, Sustain, Release)
// which characterizes the amplitude of the audio signal.
// This module keeps track of how long the instrument has
// been enabled for, and applies the appropriate amplitude
// envelope at that point. An audio sample is taken in as
// the input, and the enveloped signal is given as the output.


module envelope(clock_27mhz, ready, oscillator_done,
               enb, sample, envelope_done, envelope_audio_out);

      // FSM Parameters
      parameter STAND_BY_FOR_OSCILLATOR_DONE = 8'd0;
      parameter UPDATE_ADSR = 8'd1;
      parameter COLLECT_SAMPLES = 8'd2;
      parameter ADJUST_AMPLITUDE = 8'd3;
      parameter OUTPUT_SAMPLE = 8'd4;

      // ADSR Parameters
      parameter ATTACK_STATE = 3'd0;
      parameter DECAY_STATE = 3'd1;
      parameter SUSTAIN_STATE = 3'd2;
      parameter RELEASE_STATE = 3'd3;
      parameter OFF_STATE = 3'd4;

      parameter ATTACK_DURATION = 18'd5325;
      parameter DECAY_DURATION = 18'd1229;
      parameter SUSTAIN_DURATION = 0; // Duration 0 means infinite sustain
      parameter RELEASE_DURATION = 18'd4096;

      parameter ATTACK_STEP = 1'd1; // Increment/Decrement step size per sample
      parameter DECAY_STEP = 1'd1;
      parameter SUSTAIN_STEP = 0;
      parameter RELEASE_STEP = 1'd1;

      parameter AFTER_ATTACK_HEIGHT = 18'd5325; //(ATTACK_STEP * ATTACK_DURATION)
      parameter AFTER_DECAY_HEIGHT = 18'd4096;// (AFTER_ATTACK_HEIGHT) -
(DECAY_STEP * DECAY_DURATION), ideally
                                    // the height of AMPLITUDE_RESOLUTION
      parameter AFTER_SUSTAIN_HEIGHT = 18'd4096; // (AFTER_DECAY_HEIGHT) -
(SUSTAIN_STEP * SUSTAIN_DURATION)

      parameter AMPLITUDE_RESOLUTION = 5'd12; //Power of 2, 1024-bit

      // (STEP*DURATION)/RESOLUTION = ACTUAL HEIGHT CHANGE

      input clock_27mhz;
      input ready;
      input oscillator_done;
      input enb;
      input signed [15:0] sample;
      output envelope_done;
      output signed [15:0] envelope_audio_out;

      reg envelope_done;
      reg [15:0] envelope_audio_out;
      reg [7:0] state;
      reg prev_enb;
      reg [2:0] adsr_state; // Keeps track of whether the instrument is in
                        // its Attack, Decay, Sustain, or Release
      reg [31:0] adsr_timer;// Counts up to the duration of that particular ADSR
phase
      reg signed [31:0] received_sample; // Sample received from the sample table
      reg signed [31:0] modified_sample; // Sample modified with the appropriate
amplitude envelope

      initial envelope_done <= 0;
      initial envelope_audio_out <= 0;
      initial state <= STAND_BY_FOR_OSCILLATOR_DONE;
      initial prev_enb <= 0;
      initial adsr_state <= OFF_STATE;
      initial adsr_timer <= 0;
      initial received_sample <= 0;
      initial modified_sample <= 0;

      always @ (posedge clock_27mhz) begin

         case (state)

         // Wait until the oscillator has finished calculating the address of the
next
         // sample. When that is done, move on to update the adsr_state.
         STAND_BY_FOR_OSCILLATOR_DONE:
            begin
               if (oscillator_done)
                  state <= UPDATE_ADSR;
            end

         // Update the adsr_state by looking at the instrument's current state,
whether
         // enb is on or not, and the status of the adsr_timer
         UPDATE_ADSR:
            begin
               if (~prev_enb) begin
                  if (enb) begin
                     adsr_state <= ATTACK_STATE;
                     adsr_timer <= 0;
                  end

                  else begin
                     case (adsr_state)
                        RELEASE_STATE:
                           begin
                              if (adsr_timer < RELEASE_DURATION)
                                 adsr_state <= RELEASE_STATE;
                              else
                                 adsr_state <= OFF_STATE;
                           end
                        OFF_STATE:
                           begin
                              adsr_state <= OFF_STATE;
                           end
                        default: adsr_state <= OFF_STATE;
                     endcase
                  end
               end

               else begin
                  if (~enb) begin
                     if (adsr_state == OFF_STATE)
                        adsr_state <= OFF_STATE;

                     else begin
                     adsr_state <= RELEASE_STATE;
                     adsr_timer <= 0;
                     end
                  end

                  else begin
                     case (adsr_state)
                        ATTACK_STATE:
                           begin
                              if (adsr_timer < ATTACK_DURATION)
                                 adsr_state <= ATTACK_STATE;
                              else begin
                                 adsr_state <= DECAY_STATE;
                                 adsr_timer <= 0;
                              end
                           end
                        DECAY_STATE:
                           begin
                              if (adsr_timer < DECAY_DURATION)
                                 adsr_state <= DECAY_STATE;
                              else begin
                                 adsr_state <= SUSTAIN_STATE;
                                 adsr_timer <= 0;
                              end
                           end
                        SUSTAIN_STATE:
                           begin
                              if (SUSTAIN_DURATION == 0) begin
                                 adsr_state <= SUSTAIN_STATE;
                                 adsr_timer <= 0;
                              end

                              else begin
                                 if (adsr_timer < SUSTAIN_DURATION)
                                    adsr_state <= SUSTAIN_STATE;
                                 else begin
                                    adsr_state <= RELEASE_STATE;
                                    adsr_timer <= 0;
                                 end
```

```
                              end
                           end
                        RELEASE_STATE:
                           begin
                              if (adsr_timer < RELEASE_DURATION)
                                 adsr_state <= RELEASE_STATE;
                              else begin
                                 adsr_state <= OFF_STATE;
                                 adsr_timer <= 0;
                              end
                           end
                        OFF_STATE:
                           begin
                              adsr_state <= OFF_STATE;
                           end
                        default: adsr_state <= OFF_STATE;
                     endcase
                  end
               end

               prev_enb <= enb;
               state <= COLLECT_SAMPLES;
            end

            // Obtain the audio sample at the address specified by the oscillator
            COLLECT_SAMPLES:
               begin
                  received_sample <= sample;
                  state <= ADJUST_AMPLITUDE;
               end

            // Modify the amplitude of the obtained sample according to the
            // calculated adsr paramters
            ADJUST_AMPLITUDE:
               begin
                  case (adsr_state)
                     ATTACK_STATE:
                        begin
                           modified_sample <= (((ATTACK_STEP*adsr_timer)
                                              *received_sample) >>
AMPLITUDE_RESOLUTION);
                           adsr_timer <= adsr_timer + 1;
                           state <= OUTPUT_SAMPLE;
                        end
                     DECAY_STATE:
                        begin
                           modified_sample <= (((AFTER_ATTACK_HEIGHT -
(DECAY_STEP*adsr_timer))
                                                *received_sample) >>
AMPLITUDE_RESOLUTION);
                           adsr_timer <= adsr_timer + 1;
                           state <= OUTPUT_SAMPLE;
                        end
                     SUSTAIN_STATE:
                        begin
                           if (SUSTAIN_DURATION == 0) begin
                              modified_sample <= ((AFTER_DECAY_HEIGHT *
received_sample)
                                          >> AMPLITUDE_RESOLUTION);
                              state <= OUTPUT_SAMPLE;
                           end
                           else begin
                              modified_sample <= (((AFTER_DECAY_HEIGHT -
(SUSTAIN_STEP*adsr_timer))
                                          *received_sample) >>
AMPLITUDE_RESOLUTION);
                              adsr_timer <= adsr_timer + 1;
                              state <= OUTPUT_SAMPLE;
                           end
                        end
                     RELEASE_STATE:
                        begin
                           modified_sample <= (((AFTER_SUSTAIN_HEIGHT -
(RELEASE_STEP*adsr_timer))
                                       *received_sample) >>
AMPLITUDE_RESOLUTION);
                           adsr_timer <= adsr_timer + 1;
                           state <= OUTPUT_SAMPLE;
                        end
                     OFF_STATE:
                        begin
                           modified_sample <= 0;
                           state <= OUTPUT_SAMPLE;
                        end
                     default: adsr_state <= OFF_STATE;
                  endcase
               end

            // output the modified sample as an audio_out
            OUTPUT_SAMPLE:
               begin
                  envelope_audio_out <= modified_sample;
                  envelope_done <= 1;
                  if (ready) begin
                     envelope_done <= 0;
                     state <= STAND_BY_FOR_OSCILLATOR_DONE;
                  end
               end
            endcase
         end
endmodule
```

Appendix A: Audio Synthesis: mixer.v

```
// This is a stereo mixer that takes in all of the 16-bit audio_out
// signals given by the instrument managers, add them up into a
// 24-bit register, and scale it to fit into a 16-bit stereo output.

module mixer(clock_27mhz,

            // Strings
            violin1_audio_out, violin2_audio_out, viola_audio_out,
            cello_audio_out, bass_audio_out,
            // Brass
            trumpet1_audio_out, trumpet2_audio_out, horn_audio_out,
            trombone_audio_out, tuba_audio_out,

            // Woodwinds
            flute_audio_out, clarinet_audio_out, oboe_audio_out,
            sax_audio_out, basoon_audio_out,

            // Percussions
            timpani_audio_out, snare_audio_out, hihat_audio_out,
            crash_audio_out, bassdrum_audio_out,

            audio_out_left, audio_out_right);

      input clock_27mhz;

      input signed [15:0] violin1_audio_out, violin2_audio_out, viola_audio_out,
                     cello_audio_out, bass_audio_out;

      input signed [15:0] trumpet1_audio_out, trumpet2_audio_out, horn_audio_out,
                     trombone_audio_out, tuba_audio_out;

      input signed [15:0] flute_audio_out, clarinet_audio_out, oboe_audio_out,
                     sax_audio_out, basoon_audio_out;

      input signed [15:0] timpani_audio_out, snare_audio_out, hihat_audio_out,
                     crash_audio_out, bassdrum_audio_out;

      output signed [15:0] audio_out_left, audio_out_right;

      reg signed [15:0] audio_out_left, audio_out_right;

      // A temporary 24-bit register to store the added signals
      reg signed [23:0] temp_left_out, temp_right_out;

      reg [4:0] state;

      initial temp_left_out <= 0;
      initial temp_right_out <= 0;
      initial state <= 0;

      // All of the signals cannot be added in one clock cycle, as it
      // would not meet the timing requirements for our 27MHz clock.
      // Instead, since there are extra clock cycles in between the
      // ready signals from the DAC, the signals are added one
      // by one per clock cycle.
      always @ (posedge clock_27mhz) begin

         case (state)
            5'd0:
               begin
               temp_left_out <= violin1_audio_out + violin2_audio_out;
               temp_right_out <= violin1_audio_out + violin2_audio_out;
               state <= 5'd1;
               end
            5'd1:
               begin
               temp_left_out <= temp_left_out + viola_audio_out;
               temp_right_out <= temp_right_out + viola_audio_out;
               state <= 5'd2;
               end
            5'd2:
               begin
               temp_left_out <= temp_left_out + cello_audio_out;
               temp_right_out <= temp_right_out + cello_audio_out;
               state <= 5'd3;
               end
            5'd3:
               begin
               temp_left_out <= temp_left_out + bass_audio_out;
               temp_right_out <= temp_right_out + bass_audio_out;
               state <= 5'd4;
               end
            5'd4:
               begin
               temp_left_out <= temp_left_out + trumpet1_audio_out;
               temp_right_out <= temp_right_out + trumpet1_audio_out;
               state <= 5'd5;
               end
            5'd5:
               begin
               temp_left_out <= temp_left_out + trumpet2_audio_out;
               temp_right_out <= temp_right_out + trumpet2_audio_out;
               state <= 5'd6;
               end
            5'd6:
               begin
               temp_left_out <= temp_left_out + horn_audio_out;
               temp_right_out <= temp_right_out + horn_audio_out;
               state <= 5'd7;
               end
            5'd7:
               begin
               temp_left_out <= temp_left_out + trombone_audio_out;
               temp_right_out <= temp_right_out + trombone_audio_out;
               state <= 5'd8;
               end
            5'd8:
               begin
               temp_left_out <= temp_left_out + tuba_audio_out;
               temp_right_out <= temp_right_out + tuba_audio_out;
               state <= 5'd9;
               end
            5'd9:
               begin
               temp_left_out <= temp_left_out + flute_audio_out;
               temp_right_out <= temp_right_out + flute_audio_out;
               state <= 5'd10;
               end
            5'd10:
               begin
               temp_left_out <= temp_left_out + clarinet_audio_out;
               temp_right_out <= temp_right_out + clarinet_audio_out;
               state <= 5'd11;
               end
            5'd11:
               begin
               temp_left_out <= temp_left_out + oboe_audio_out;
               temp_right_out <= temp_right_out + oboe_audio_out;
               state <= 5'd12;
               end
            5'd12:
               begin
               temp_left_out <= temp_left_out + sax_audio_out;
               temp_right_out <= temp_right_out + sax_audio_out;
               state <= 5'd13;
               end
            5'd13:
               begin
               temp_left_out <= temp_left_out + basoon_audio_out;
               temp_right_out <= temp_right_out + basoon_audio_out;
               state <= 5'd14;
               end
            5'd14:
               begin
               temp_left_out <= temp_left_out + timpani_audio_out;
               temp_right_out <= temp_right_out + timpani_audio_out;
               state <= 5'd15;
               end
            5'd15:
               begin
               temp_left_out <= temp_left_out + ((10 * snare_audio_out) >> 3);
               temp_right_out <= temp_right_out + ((10 * snare_audio_out) >> 3);
               state <= 5'd16;
               end
            5'd16:
               begin
               temp_left_out <= temp_left_out + hihat_audio_out;
               temp_right_out <= temp_right_out + hihat_audio_out;
               state <= 5'd17;
               end
            5'd17:
               begin
               temp_left_out <= temp_left_out + crash_audio_out;
               temp_right_out <= temp_right_out + crash_audio_out;
               state <= 5'd18;
               end
            5'd18:
               begin
               temp_left_out <= temp_left_out + bassdrum_audio_out;
               temp_right_out <= temp_right_out + bassdrum_audio_out;
               state <= 5'd19;
               end
            5'd19:
               begin
               temp_left_out <= ((5 * temp_left_out) >> 3);
               temp_right_out <= ((5 * temp_right_out) >> 3);
               state <= 5'd20;
               end
            5'd20:
               begin
```

```
               audio_out_left <= temp_left_out[15:0];
               audio_out_right <= temp_right_out[15:0];
               state <= 5'd0;
               end
            default:
               state <= 5'd0;
         endcase
      end
endmodule
```

Appendix A: Audio Synthesis: orchestra.v

```
// Top-level module to be instantiated in the labkit. This module encapsulates
// the entire sound synthesis block of the project to allow smooth integration
// with the rest of the project. This module also makes the sound synthesis
// block very modular, allowing changes to happen inside the sound synthesis
// block without the need for changes in the labkit or any other external
// modules.
//
// It takes in the enables and pitch values from the Score Manager, and the
// volume value from the conductor, and outputs the appropriate audio signals
// to the AC97.

module orchestra(clock_27mhz, reset, volume_from_conductor,

               // DAC Parameters
               audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
ac97_bit_clock,

               // Instrument Parameters

               // Strings
               violin1_enb, violin2_enb, viola_enb, cello_enb, bass_enb,
               violin1_pitch, violin2_pitch, viola_pitch, cello_pitch, bass_pitch,

               // Brass
               trumpet1_enb, trumpet2_enb, horn_enb, trombone_enb, tuba_enb,
               trumpet1_pitch, trumpet2_pitch, horn_pitch, trombone_pitch,
tuba_pitch,

               // Woodwinds
               flute_enb, clarinet_enb, oboe_enb, sax_enb, basoon_enb,
               flute_pitch, clarinet_pitch, oboe_pitch, sax_pitch, basoon_pitch,

               // Percussion
               timpani_enb, snare_enb, hihat_enb, crash_enb, bassdrum_enb,
               timpani_pitch);


      input clock_27mhz;
      input reset;

      // DAC Parameters
      //-------------------------------------------------------
      output audio_reset_b;
         output ac97_sdata_out;
         input ac97_sdata_in;
         output ac97_synch;
         input ac97_bit_clock;

         input [3:0] volume_from_conductor; // volume value received from
                                 // the conductor module

      wire ready; // a 48kHz square wave generated by the DAC to request a new
sample
      wire [4:0] volume; // volume value to be sent to the DAC


      // Pulse Generator Parameters
      //-------------------------------------------------------
      wire ready_pulse; // A pulse version of the ready signal generated by the DAC


      // Instrument Manager Parameters
      //-------------------------------------------------------
      // Each instrument has an enable, a pitch, and an audio_out.
      // When enable is low, the instrument is off. When enable is high,
      // the audio signal of the instrument is sent to its audio_out at
      // the pitch specified by the instrument's pitch value.

      // Strings
      input violin1_enb, violin2_enb, viola_enb, cello_enb, bass_enb;
      input [6:0] violin1_pitch, violin2_pitch, viola_pitch, cello_pitch,
bass_pitch;
      wire signed [15:0] violin1_audio_out, violin2_audio_out, viola_audio_out,
                     cello_audio_out, bass_audio_out;

      // Brass
      input trumpet1_enb, trumpet2_enb, horn_enb, trombone_enb, tuba_enb;
      input [6:0] trumpet1_pitch, trumpet2_pitch, horn_pitch, trombone_pitch,
tuba_pitch;
      wire signed [15:0] trumpet1_audio_out, trumpet2_audio_out, horn_audio_out,
                     trombone_audio_out, tuba_audio_out;

      // Woodwinds
      input flute_enb, clarinet_enb, oboe_enb, sax_enb, basoon_enb;
      input [6:0] flute_pitch, clarinet_pitch, oboe_pitch, sax_pitch, basoon_pitch;
      wire signed [15:0] flute_audio_out, clarinet_audio_out, oboe_audio_out,
                     sax_audio_out, basoon_audio_out;

      // Percussion
      input timpani_enb, snare_enb, hihat_enb, crash_enb, bassdrum_enb;
      input [6:0] timpani_pitch;
      wire signed [15:0] timpani_audio_out, snare_audio_out, hihat_audio_out,
                     crash_audio_out, bassdrum_audio_out;


      // Mixer Parameters
      //-------------------------------------------------------
      wire signed [15:0] audio_out_left; // left-channel audio to be sent to the
DAC
      wire signed [15:0] audio_out_right;// right-channel audio to be sent to the
DAC
```

```
	// Instantiated Modules

	// DAC - The DAC requests a sample by sending a ready signal.
	//------------------------------------------------------
	assign volume = {1'b1, volume_from_conductor}; // Assigns the output volume 
specified
	                                    // by the conductor

	dac dac_driver(clock_27mhz, reset, volume, audio_out_left, audio_out_right, 
ready,
	               audio_reset_b, ac97_sdata_out, ac97_sdata_in,
	               ac97_synch, ac97_bit_clock);

	// Pulse Generator - The ready signal from the DAC is converted into a pulse.
	//------------------------------------------------------

	pulse_generator pulse(clock_27mhz, ready, ready_pulse);


	// Instrument Managers - Every time a ready_pulse is asserted, the instrument 
managers
	//                 output the audio_out of each of the instruments they are
	//                 responsible for.
	//------------------------------------------------------

	strings_manager strings(clock_27mhz, ready_pulse,
	                  violin1_enb, violin2_enb, viola_enb, cello_enb, bass_enb,
	                  violin1_pitch, violin2_pitch, viola_pitch, cello_pitch, 
bass_pitch,
	                  violin1_audio_out, violin2_audio_out, viola_audio_out,
	                  cello_audio_out, bass_audio_out);

	brass_manager brass(clock_27mhz, ready_pulse,
	               trumpet1_enb, trumpet2_enb, horn_enb, trombone_enb, tuba_enb,
	               trumpet1_pitch, trumpet2_pitch, horn_pitch, trombone_pitch, 
tuba_pitch,
	               trumpet1_audio_out, trumpet2_audio_out, horn_audio_out,
	               trombone_audio_out, tuba_audio_out);

	woodwinds_manager woodwinds(clock_27mhz, ready_pulse,
	                     flute_enb, clarinet_enb, oboe_enb, sax_enb, basoon_enb,
	                     flute_pitch, clarinet_pitch, oboe_pitch, sax_pitch, 
basoon_pitch,
	                     flute_audio_out, clarinet_audio_out, oboe_audio_out,
	                     sax_audio_out, basoon_audio_out);

	percussion_manager percussion(clock_27mhz, ready_pulse,
	                       timpani_enb, snare_enb, hihat_enb, crash_enb, 
bassdrum_enb,
	                       timpani_pitch,
	                       timpani_audio_out, snare_audio_out, hihat_audio_out,
	                       crash_audio_out, bassdrum_audio_out);


	// Mixer - The audio_outs from the instrument managers are mixed into single
```

```
left
      //          and right channels to be sent to the DAC.
      //-----------------------------------------------------

      mixer stereomixer(clock_27mhz,
                     // Strings
                     violin1_audio_out, violin2_audio_out, viola_audio_out,
                     cello_audio_out, bass_audio_out,

                     // Brass
                     trumpet1_audio_out, trumpet2_audio_out, horn_audio_out,
                     trombone_audio_out, tuba_audio_out,

                     // Woodwinds
                     flute_audio_out, clarinet_audio_out, oboe_audio_out,
                     sax_audio_out, basoon_audio_out,

                     // Percussions
                     timpani_audio_out, snare_audio_out, hihat_audio_out,
                     crash_audio_out, bassdrum_audio_out,

                     // Stereo Audio Out
                     audio_out_left, audio_out_right);


endmodule
```

Appendix A: Audio Synthesis: oscillator.v

```
// This module is the skeleton of all oscillator modules.
// Depending on the instrument, the original frequency at
// which the sample was recorded changes, and therefore the
// period table used in this module would be different. Hence
// all of the oscillator modules (string_oscillator,
// brass_oscillator, etc) are versions of this module with
// different period table BRAMs.
//
// This module takes in a pitch, and calculates the address
// of the next sample that corresponds with this pitch.

module oscillator(clock_27mhz, ready, pitch, sample_table_size, envelope_done,
addr, oscillator_done);
   parameter READY_SIGNAL_PERIOD = 10'd563;

   // FSM Parameters
   parameter STAND_BY_FOR_READY = 2'd0;
   parameter STAND_BY_FOR_ENVELOPE_DONE = 2'd1;
   parameter CALCULATE_ADDRESS = 2'd2;

   input clock_27mhz;
   input ready;
   input [6:0] pitch;
   input [15:0] sample_table_size; // Size of the sample table BRAM
   input envelope_done;
   output [15:0] addr;
   output oscillator_done;
   reg [15:0] addr;
   reg oscillator_done;
   reg [31:0] remainder;
   reg [1:0] state;

   wire [16:0] period;

   // Period table BRAM - list of period values for different pitches
   period_table_60 period_table(pitch, clock_27mhz, period);

   initial addr <= 0;
   initial oscillator_done <= 0;
   initial remainder <= 32'd563;
   initial state <= STAND_BY_FOR_READY;


   always @ (posedge clock_27mhz) begin

   case (state)

      // Wait until the next ready pulse
      STAND_BY_FOR_READY:
         begin
            if (ready) begin
               oscillator_done <= 0;
               state <= STAND_BY_FOR_ENVELOPE_DONE;
            end
         end

      // Wait until the previous instrument's envelope is done
      // (brass and strings only. There will be no wait here in
      // the case of woodwinds, as the calculation of the audio
      // signals is simultaneous, not sequential).
      STAND_BY_FOR_ENVELOPE_DONE:
         begin
            if (envelope_done) state <= CALCULATE_ADDRESS;
         end

      // Given the pitch, calculate the next address of the sample
      // by looking at how many times the address must have
      // incremented between the last and the next ready pulse.
      CALCULATE_ADDRESS:
         begin
            if (remainder >= period) begin
               remainder <= remainder - period;
               if (addr == sample_table_size) addr <= 0;
               else addr <= addr + 1;
            end
            else begin
               remainder <= remainder + READY_SIGNAL_PERIOD;
               oscillator_done <= 1;
               state <= STAND_BY_FOR_READY;
            end
         end
      endcase
   end
endmodule
```

Appendix A: Audio Synthesis: percussion_manager.v

```
// An instrument manager module responsible for the
// percussion section. It takes in the enables for
// timpani, snare, hihat, crash, and bass drum, and the
// pitch for the timpani, and produces the corresponding
// audio_outs.

module percussion_manager(clock_27mhz, ready,

                     // Enables
                     timpani_enb, snare_enb, hihat_enb, crash_enb, bassdrum_enb,

                     // Pitches
                     timpani_pitch,

                     // Audio Outs
                     timpani_audio_out, snare_audio_out, hihat_audio_out,
crash_audio_out, bassdrum_audio_out);

   // Size of the sample tables
   parameter TIMPANI_SAMPLE_TABLE_SIZE = 16'd12428;
   parameter SNARE_SAMPLE_TABLE_SIZE = 16'd10647;
   parameter HIHAT_SAMPLE_TABLE_SIZE = 16'd10412;
   parameter CRASH_SAMPLE_TABLE_SIZE = 16'd19672;
   parameter BASSDRUM_SAMPLE_TABLE_SIZE = 16'd4711;

   input clock_27mhz;
   input ready;

   input timpani_enb, snare_enb, hihat_enb, crash_enb, bassdrum_enb;
   input [6:0] timpani_pitch;

   output signed [15:0] timpani_audio_out, snare_audio_out, hihat_audio_out,
                crash_audio_out, bassdrum_audio_out;

   reg signed [15:0] timpani_audio_out, snare_audio_out, hihat_audio_out,
               crash_audio_out, bassdrum_audio_out;

   // The percussion audio signals differ from the rest of the orchestra
   // instruments in a sense that it is not periodic, and it must trigger
   // on the rising edge of the enb signal and decay even if enb is turned
   // low. The audio samples already have their ADSR envelopes applied to
   // them as well, so the oscillator and the envelope modules cannot be
   // used here. Instead, each instrument will be played back at their
   // recorded sample rate, at the rising edge of the enb signal. In the
   // case of the timpani, the sample rate would depend on its pitch.


   // Registers to keep track of the previous state of enb.
   reg prev_timpani_enb, prev_snare_enb, prev_hihat_enb, prev_crash_enb,
prev_bassdrum_enb;

   // Counters to determine the sampling rate.
   reg [11:0] timpani_counter, snare_counter, hihat_counter, crash_counter,
bassdrum_counter;
```

```
   wire [19:0] timpani_period; // Period of timpani signal given from the period
table

   // Sample table addresses for each instrument.
   reg [15:0] timpani_addr, snare_addr, hihat_addr, crash_addr, bassdrum_addr;

   // Sample from the sample table for each instrument.
   wire signed [15:0] timpani_audio_sample, snare_audio_sample, hihat_audio_sample,
crash_audio_sample,
                  bassdrum_audio_sample;


   // Audio sample table BRAMs
   new_snare snare(snare_addr,clock_27mhz,snare_audio_sample);
   new_hihat hihat(hihat_addr, clock_27mhz, hihat_audio_sample);
   new_crash crash(crash_addr, clock_27mhz, crash_audio_sample);
   new_bassdrum bassdrum(bassdrum_addr, clock_27mhz, bassdrum_audio_sample);
   new_timp timpani(timpani_addr, clock_27mhz, timpani_audio_sample);

   // Period table BRAM to determine the sampling rate of timpani.
   period_table_60_timp period_table(timpani_pitch, clock_27mhz, timpani_period);

   initial snare_addr <= 0;
   initial hihat_addr <= 0;
   initial crash_addr <= 0;
   initial bassdrum_addr <= 0;
   initial timpani_addr <= 0;

   // Each of the following always blocks has a counter that counts up to the
   // sampling period of each instrument, outputs the audio sample at the
   // current address, and increment the address. For all instruments other
   // than the timpani, the entire sample table is played at every positive
   // edge of the enb signal.

   // Calculate audio signal for timpani
   always @ (posedge clock_27mhz) begin

      if (timpani_counter == timpani_period) begin

         if (timpani_enb) begin
            if (timpani_addr == TIMPANI_SAMPLE_TABLE_SIZE) begin
               timpani_addr <= TIMPANI_SAMPLE_TABLE_SIZE;
               timpani_audio_out <= 0;
            end

            else begin
               timpani_audio_out <= timpani_audio_sample;
               timpani_addr <= timpani_addr + 1;
            end
         end

         else timpani_addr <= 0;

         timpani_counter <= 0;
      end
      else timpani_counter <= timpani_counter + 1;

   end


   // Calculate the audio signal for snare
   always @ (posedge clock_27mhz) begin

      if (snare_counter == 612) begin

         if (snare_enb) begin

            if (snare_enb == !prev_snare_enb) begin
               snare_addr <= 0;
            end

            else if (snare_addr == SNARE_SAMPLE_TABLE_SIZE)
               snare_addr <= SNARE_SAMPLE_TABLE_SIZE;

            else begin
               snare_audio_out <= snare_audio_sample;
               snare_addr <= snare_addr + 1;
            end
         end

         else begin
            if (snare_addr == SNARE_SAMPLE_TABLE_SIZE)
               snare_addr <= SNARE_SAMPLE_TABLE_SIZE;
            else begin
               snare_audio_out <= snare_audio_sample;
               snare_addr <= snare_addr + 1;
            end
         end
         prev_snare_enb <= snare_enb;
         snare_counter <= 0;
      end

      else snare_counter <= snare_counter + 1;

   end


   // Calculate the audio signal for crash
   always @ (posedge clock_27mhz) begin

      if (crash_counter == 612) begin

         if (crash_enb) begin

            if (crash_enb == !prev_crash_enb) begin
               crash_addr <= 0;
            end

            else if (crash_addr == CRASH_SAMPLE_TABLE_SIZE)
               crash_addr <= CRASH_SAMPLE_TABLE_SIZE;

            else begin
```

```
            crash_audio_out <= crash_audio_sample;
            crash_addr <= crash_addr + 1;
         end
      end

      else begin
         if (crash_addr == CRASH_SAMPLE_TABLE_SIZE)
            crash_addr <= CRASH_SAMPLE_TABLE_SIZE;
         else begin
            crash_audio_out <= crash_audio_sample;
            crash_addr <= crash_addr + 1;
         end
      end
      prev_crash_enb <= crash_enb;
      crash_counter <= 0;
   end

   else crash_counter <= crash_counter + 1;

end

// Calculate the audio signal for hihat
always @ (posedge clock_27mhz) begin

   if (hihat_counter == 612) begin

      if (hihat_enb) begin

         if (hihat_enb == !prev_hihat_enb) begin
            hihat_addr <= 0;
         end

         else if (hihat_addr == HIHAT_SAMPLE_TABLE_SIZE)
            hihat_addr <= HIHAT_SAMPLE_TABLE_SIZE;

         else begin
            hihat_audio_out <= hihat_audio_sample;
            hihat_addr <= hihat_addr + 1;
         end
      end

      else begin
         if (hihat_addr == HIHAT_SAMPLE_TABLE_SIZE)
            hihat_addr <= HIHAT_SAMPLE_TABLE_SIZE;
         else begin
            hihat_audio_out <= hihat_audio_sample;
            hihat_addr <= hihat_addr + 1;
         end
      end
      prev_hihat_enb <= hihat_enb;
      hihat_counter <= 0;
   end

   else hihat_counter <= hihat_counter + 1;

end
```

```
   // Calculate the audio signal for bassdrum
   always @ (posedge clock_27mhz) begin

      if (bassdrum_counter == 612) begin

         if (bassdrum_enb) begin

            if (bassdrum_enb == !prev_bassdrum_enb) begin
               bassdrum_addr <= 0;
            end

            else if (bassdrum_addr == BASSDRUM_SAMPLE_TABLE_SIZE)
               bassdrum_addr <= BASSDRUM_SAMPLE_TABLE_SIZE;

            else begin
               bassdrum_audio_out <= bassdrum_audio_sample;
               bassdrum_addr <= bassdrum_addr + 1;
            end
         end

         else begin
            if (bassdrum_addr == BASSDRUM_SAMPLE_TABLE_SIZE)
               bassdrum_addr <= BASSDRUM_SAMPLE_TABLE_SIZE;
            else begin
               bassdrum_audio_out <= bassdrum_audio_sample;
               bassdrum_addr <= bassdrum_addr + 1;
            end
         end
         prev_bassdrum_enb <= bassdrum_enb;
         bassdrum_counter <= 0;
      end

      else bassdrum_counter <= bassdrum_counter + 1;

   end

endmodule
```

Appendix A: Audio Synthesis: pulse_generator.v

```
// Given the ready signal from the DAC, this module
// converts it into a pulse at the rising edge of the
// signal.

module pulse_generator(clock_27mhz, ready, ready_pulse);

      input clock_27mhz;
      input ready;
      output ready_pulse;

      reg ready_pulse;
      reg last_ready;

      initial ready_pulse <= 0;
      initial last_ready <= 0;
      always @ (posedge clock_27mhz) begin
         if(ready) begin
            if(ready == ~last_ready)
               ready_pulse <= 1;
            else ready_pulse <= 0;
            last_ready <= ready;
         end
         else begin
            last_ready <= ready;
            ready_pulse <= 0;
         end
      end
endmodule
```

Appendix A: Audio Synthesis: strings_manager.v

```
// An instrument manager module responsible for the
// strings section. It takes in the enables and pitches
// for violin1, violin2, viola, cello, and bass,
// and produces the corresponding audio_outs.

module strings_manager(clock_27mhz, ready,

                        // Enables
                        violin1_enb, violin2_enb, viola_enb, cello_enb, bass_enb,

                        // Pitches
                        violin1_pitch, violin2_pitch, viola_pitch, cello_pitch,
bass_pitch,

                        // Audio Outs
                        violin1_audio_out, violin2_audio_out, viola_audio_out,
                        cello_audio_out, bass_audio_out

                        );

   parameter SAMPLE_TABLE_SIZE = 16'd46751; // Size of the string sample table

   input clock_27mhz;
   input ready;

   input violin1_enb, violin2_enb, viola_enb, cello_enb, bass_enb;
   input [6:0] violin1_pitch, violin2_pitch, viola_pitch, cello_pitch, bass_pitch;

   output signed [15:0] violin1_audio_out, violin2_audio_out, viola_audio_out,
                   cello_audio_out, bass_audio_out;

   reg [15:0] sample_addr; // variable that selects the address of the string
                     // sample table


   // Since the DAC samples at 48kHz while our clock runs at 27MHz, there are
   // about 563 clock cycles in between each sample. Therefore the calculation
   // of the audio signal for each instrument is done sequentially, sharing the
   // same sample table among the instruments, and calculating the audio signals
   // instrument by instrument within the 563 clock cycles that are available.
   wire [4:0] progress; // Indicates which instrument's audio signals have been
                   // calculated.

   // The address for the sample table for each instrument.
   wire [15:0] violin1_addr, violin2_addr, viola_addr, cello_addr, bass_addr;

   wire signed [15:0] sample; // The 16-bit sample out of the sample table


   // These "done" signals show that the instrument has finished calculating its
   // audio signal
   wire violin1_oscillator_done, violin1_envelope_done;
   wire violin2_oscillator_done, violin2_envelope_done;
   wire viola_oscillator_done, viola_envelope_done;
   wire cello_oscillator_done, cello_envelope_done;
   wire bass_oscillator_done, bass_envelope_done;


   assign progress[0] = 1'b1; // Violin1 is calculating its audio signal.
   assign progress[1] = violin1_envelope_done; // Violin2 is calculating its audio
signal.
   assign progress[2] = violin2_envelope_done; // Viola is calculating its audio
signal.
   assign progress[3] = viola_envelope_done; // Cello is calculating its audio
signal.
   assign progress[4] = cello_envelope_done; // Bass is calculating its audio
signal.


   // String Sample Table BRAM - Contains the string audio samples
   strings_sample strings(sample_addr,clock_27mhz,sample);


   // Audio-Signal Calculation - These modules sequentially calculate the audio
signal
   //                      for each instrument. The string_oscillator calculates
   //                         the address of the sample, given the pitch of the
   //                      instrument. Then, the envelope module takes the sample
   //                      from the sample table at the address specified by
   //                      the string_oscillator, applies an ADSR envelope to the
   //                      obtained sample, and outputs the resulting signal
   //                      as the audio_out for the instrument.
   //---------------------------------------------------------------------------------
----


   // Violin 1
   string_oscillator violin1_osc(clock_27mhz, ready, violin1_pitch,
SAMPLE_TABLE_SIZE,
                           1'b1, violin1_addr, violin1_oscillator_done);

   envelope violin1_env(clock_27mhz, ready, violin1_oscillator_done, violin1_enb,
                  sample, violin1_envelope_done, violin1_audio_out);

   // Violin 2
   string_oscillator violin2_osc(clock_27mhz, ready, violin2_pitch,
SAMPLE_TABLE_SIZE,
                           violin1_envelope_done, violin2_addr,
violin2_oscillator_done);

   envelope violin2_env(clock_27mhz, ready, violin2_oscillator_done, violin2_enb,
                  sample, violin2_envelope_done, violin2_audio_out);

   // Viola
   string_oscillator viola_osc(clock_27mhz, ready, viola_pitch, SAMPLE_TABLE_SIZE,
                           violin2_envelope_done, viola_addr,
viola_oscillator_done);

   envelope viola_env(clock_27mhz, ready, viola_oscillator_done, viola_enb,
                  sample, viola_envelope_done, viola_audio_out);

   // Cello
   string_oscillator cello_osc(clock_27mhz, ready, cello_pitch, SAMPLE_TABLE_SIZE,
                           viola_envelope_done, cello_addr, cello_oscillator_done);

   envelope cello_env(clock_27mhz, ready, cello_oscillator_done, cello_enb,
                  sample, cello_envelope_done, cello_audio_out);

   // Bass
   string_oscillator bass_osc(clock_27mhz, ready, bass_pitch, SAMPLE_TABLE_SIZE,
                           cello_envelope_done, bass_addr, bass_oscillator_done);

   envelope bass_env(clock_27mhz, ready, bass_oscillator_done, bass_enb,
                  sample, bass_envelope_done, bass_audio_out);



   // This always loop sequentially assigns the address given by the oscillator
   // as the address to the sample table.
   always @ (posedge clock_27mhz) begin

      case(progress)
      // Assign the sample address as specified by the violin1 oscillator
      5'b00001:   begin
                  sample_addr <= violin1_addr;
               end

      // Assign the sample address as specified by the violin2 oscillator
      5'b00011:   begin
                  sample_addr <= violin2_addr;
               end

      // Assign the sample address as specified by the viola oscillator
      5'b00111:   begin
                  sample_addr <= viola_addr;
               end

      // Assign the sample address as specified by the cello oscillator
      5'b01111:   begin
                  sample_addr <= cello_addr;
               end

      // Assign the sample address as specified by the bass oscillator
      5'b11111:   begin
```

```
                  sample_addr <= bass_addr;
               end

         default: sample_addr <= violin1_addr;
         endcase


   end


endmodule
```

Appendix A: Audio Synthesis: woodwinds_manager.v

```
// An instrument manager module responsible for the
// woodwind section. It takes in the enables and pitches
// for flute, clarinet, oboe, sax, and basoon,
// and produces the corresponding audio_outs.

module woodwinds_manager(clock_27mhz, ready,

                        // Enables
                        flute_enb, clarinet_enb, oboe_enb, sax_enb, basoon_enb,

                        // Pitches
                        flute_pitch, clarinet_pitch, oboe_pitch, sax_pitch,
basoon_pitch,

                        // Audio Outs
                        flute_audio_out, clarinet_audio_out, oboe_audio_out,
                        sax_audio_out, basoon_audio_out);

   // Size of the sample tables
   parameter FLUTE_SAMPLE_TABLE_SIZE = 16'd7419;
   parameter CLARINET_SAMPLE_TABLE_SIZE = 16'd1029;
   parameter OBOE_SAMPLE_TABLE_SIZE = 16'd9530;
   parameter SAX_SAMPLE_TABLE_SIZE = 16'd8050;
   parameter BASOON_SAMPLE_TABLE_SIZE = 16'd2957;

   input clock_27mhz;
   input ready;

   input flute_enb, clarinet_enb, oboe_enb, sax_enb, basoon_enb;
   input [6:0] flute_pitch, clarinet_pitch, oboe_pitch, sax_pitch, basoon_pitch;
   output signed [15:0] flute_audio_out, clarinet_audio_out, oboe_audio_out,
                      sax_audio_out, basoon_audio_out;


   wire [15:0] flute_addr, clarinet_addr, oboe_addr, sax_addr, basoon_addr;
   wire signed [15:0] flute_audio_sample, clarinet_audio_sample, oboe_audio_sample,
                      sax_audio_sample, basoon_audio_sample;


   // These "done" signals show that the instrument has finished calculating its
   // audio signal
   wire flute_oscillator_done, flute_envelope_done;
   wire clarinet_oscillator_done, clarinet_envelope_done;
   wire oboe_oscillator_done, oboe_envelope_done;
   wire sax_oscillator_done, sax_envelope_done;
   wire basoon_oscillator_done, basoon_envelope_done;

   // Because the waveform of each of the woodwinds is unique, the same
   // sample table cannot be shared among the 5 woodwind instruments.
   // Therefore, unlike the strings or brass managers, 5 BRAMs are generated
   // for 5 woodwinds, and the calculation of each audio signal take place
   // simultaneously.

   // Sample table BRAMs for each instrument
   flute_sample flute(flute_addr, clock_27mhz, flute_audio_sample);
   clarinet_sample clarinet(clarinet_addr, clock_27mhz, clarinet_audio_sample);
   oboe_sample oboe(oboe_addr, clock_27mhz, oboe_audio_sample);
   sax_sample sax(sax_addr, clock_27mhz, sax_audio_sample);
   basoon_sample basoon(basoon_addr, clock_27mhz, basoon_audio_sample);


   // Audio-Signal Calculation - These modules simultaneously calculate the audio
signal
   //                      for each instrument. Each oscillator calculates
   //                         the address of the sample, given the pitch of the
   //                      instrument. Then, the envelope module takes the sample
   //                      from the sample table at the address specified by
   //                      the oscillator, applies an ADSR envelope to the
   //                      obtained sample, and outputs the resulting signal
   //                      as the audio_out for the instrument.
   //-------------------------------------------------------------------------------
----


   // flute
   flute_oscillator flute_osc(clock_27mhz, ready, flute_pitch,
FLUTE_SAMPLE_TABLE_SIZE,
                           1'b1, flute_addr, flute_oscillator_done);

   envelope flute_env(clock_27mhz, ready, flute_oscillator_done, flute_enb,
                           flute_audio_sample, flute_envelope_done,
flute_audio_out);

   // clarinet
   clarinet_oscillator clarinet_osc(clock_27mhz, ready, clarinet_pitch,
CLARINET_SAMPLE_TABLE_SIZE,
                           1'b1, clarinet_addr, clarinet_oscillator_done);

   envelope clarinet_env(clock_27mhz, ready, clarinet_oscillator_done,
clarinet_enb,
                           clarinet_audio_sample, clarinet_envelope_done,
clarinet_audio_out);

   // oboe
   oboe_oscillator oboe_osc(clock_27mhz, ready, oboe_pitch, OBOE_SAMPLE_TABLE_SIZE,
                           1'b1, oboe_addr, oboe_oscillator_done);

   envelope oboe_env(clock_27mhz, ready, oboe_oscillator_done, oboe_enb,
                           oboe_audio_sample, oboe_envelope_done, oboe_audio_out);
   // sax
   sax_oscillator sax_osc(clock_27mhz, ready, sax_pitch, SAX_SAMPLE_TABLE_SIZE,
                           1'b1, sax_addr, sax_oscillator_done);

   envelope sax_env(clock_27mhz, ready, sax_oscillator_done, sax_enb,
                           sax_audio_sample, sax_envelope_done, sax_audio_out);

   // basoon
   basoon_oscillator basoon_osc(clock_27mhz, ready, basoon_pitch,
BASOON_SAMPLE_TABLE_SIZE,
                           sax_envelope_done, basoon_addr, basoon_oscillator_done);

   envelope basoon_env(clock_27mhz, ready, basoon_oscillator_done, basoon_enb,
                           basoon_audio_sample, basoon_envelope_done,
basoon_audio_out);

endmodule
```

Appendix B: stdafx.h

```
#pragma once

#ifndef _WIN32_WINNT    // Allow use of features specific to Windows XP or later.
#define _WIN32_WINNT 0x0501
#endif

#include <stdio.h>
#include <tchar.h>
#include <string.h>
```

Appendix B: stdafx.cpp

```
#include "stdafx.h"
```

Appendix B: WriteToUSB.h

```
#include "stdafx.h"
#include "NoteData.h"
#include "../FTD2xx/ftd2xx.h"    //USB drivers
#include "FileData.h"            //Rogus MIDI parsing
#include "MidiMsg.h"
#include "NoteCollection.h"         //Our Note representers
#include "NoteData.h"
#include <algorithm>

//my stuff
void ParseMidi();
void ConvertCht();
void LoadBitfile();
bool SendBitFile(FILE* file);
bool ConvertChtFile(FILE* fileCHT ,FILE* fileBIT);
bool ConvertMidiFile(char* filenameMIDI, FILE* fileCHT);

int Compare(const void *a, const void *b);      //used for sorting notes by their
time
```

Appendix B: WriteToUSB.cpp

```
// WriteToUSB.cpp : Defines the entry point for the console application.
```

```
#include "WriteToUSB.h"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   printf("****************\n");
   printf("*Conductor Hero*\n");
   printf("****************\n\n");

   char input[100];
   while(1){
      printf("\nMAIN MENU\n\n");
      printf("   [P]arse .mid to .cht\n");
      printf("   [C]onvert .cht to .bit\n");
      printf("   [L]oad .bit to Conductor Hero\n");
      printf("   [Q]uit\n\n");
      scanf_s("%s", &input, 100);
      if(strcmp(input,"P") == 0)
         ParseMidi();
      else if(strcmp(input,"C") == 0)
         ConvertCht();
      else if(strcmp(input,"L") == 0)
         LoadBitfile();
      else if(strcmp(input,"Q") == 0)
         break;
      else
         printf("Unrecognized Input (Must be capitalized)\n");
   }
   return 0;
}

void ParseMidi(){
   FILE* fileMIDI;
   FILE* fileCHT;
   char input[100];
   char filename[104];

   while(1){
      printf("\nCONVERT MIDI FILE\n\n");  //kevin miu
      printf("   [C]ancel\n");
      printf("   (anything else to load)\n");
      printf("    Filename To Read (NO extension): \n\n");
      printf("    WARNING!!!  Any .cht file with the same name as the .mid file 
will be OVERWRITTEN!\n");
      printf("    ALSO, must have .mid, not .midi extension!!!\n");
      scanf_s("%s", &input, 100);
      if(strcmp(input,"C")==0)
         return;
      else{
         strcpy_s(filename,input);
         strcat_s(filename,".mid");
         if(fopen_s(&fileMIDI,filename,"rt") != 0){
            printf("No such file %s.\n", filename);
            continue;
         }
         strcpy_s(filename,input);
         strcat_s(filename,".cht");
         if(fopen_s(&fileCHT,filename,"wb") != 0){
            printf("Error opening file %s!\n", filename);
            continue;
         }
         fclose(fileMIDI);
         strcpy_s(filename,input);
         strcat_s(filename,".mid");
         if(ConvertMidiFile(filename,fileCHT)){
            printf("%s LOADED SUCCESSFULLY!\n", input);
            return;
         }else{
            printf("ERROR LOADING %s\n", input);
            fclose(fileCHT);
         }
      }
   }
}

void ConvertCht(){
   FILE* fileCHT;
   FILE* fileBIT;
   char input[100];
   char filename[104];

   while(1){
      printf("\nCONVERT CHT FILE\n\n");   //kevin miu
      printf("   [C]ancel\n");
      printf("   (anything else to load)\n");
      printf("    Filename To Read (NO extension): \n\n");
      printf("    WARNING!!!  Any .bit file with the same name as the .cht file
will be OVERWRITTEN!\n");
      scanf_s("%s", &input, 100);
      if(strcmp(input,"C")==0)
         return;
      else{
         strcpy_s(filename,input);
         strcat_s(filename,".cht");
         if(fopen_s(&fileCHT,filename,"rt") != 0){
            printf("No such file %s.\n", filename);
            continue;
         }
         strcpy_s(filename,input);
         strcat_s(filename,".bit");
         if(fopen_s(&fileBIT,filename,"wb") != 0){
            printf("Error opening file %s!\n", filename);
            continue;
         }
         if(ConvertChtFile(fileCHT,fileBIT)){
            printf("%s LOADED SUCCESSFULLY!\n", input);
            fclose(fileCHT);
            fclose(fileBIT);
            return;
         }else{
            printf("ERROR LOADING %s\n", input);
            fclose(fileCHT);
            fclose(fileBIT);
         }
      }
   }
}

void LoadBitfile(){
   FILE* file;
   char input[100];

   while(1){
      printf("\nLOAD BIT FILE\n\n");
      printf("   [C]ancel\n");
      printf("   (anything else to load)\n");
      printf("    Filename To Read (INCLUDE extension): ");
      scanf_s("%s", &input, 100);
      if(strcmp(input,"C")==0)
         return;
      else{
         if(fopen_s(&file,input,"rb") != 0){
            printf("No such file\n");
            continue;
         }
         if(SendBitFile(file)){
            printf("%s LOADED SUCCESSFULLY!\n", input);
            fclose(file);
            return;
         }else{
            printf("ERROR LOADING %s\n", input);
            fclose(file);
         }
      }
   }
}

bool SendBitFile(FILE* file){
   FT_STATUS usbStatus;
   FT_HANDLE usbHandle;

   //opens the only USB device attached to the computer
   usbStatus = FT_Open(0, &usbHandle);

   if(usbStatus == FT_OK)
      printf("Conductor Hero USB Interface Opened Successfully!\n");
   else{
      printf("ERROR opening USB!  Code %u.\n", usbStatus);
      return false;
   }

   //Loads data from file and gives it to the USB
   DWORD written;
   int totalWritten = 0;
   unsigned char datum[6];
// char test[100];
   while(1){
      for(int i = 0; i < 6; i++){
         datum[i] = fgetc(file);
         if(feof(file)) break;
         printf("%i\t", (unsigned int)datum[i]);
      }

      //Success!
      if(feof(file)){
         printf("   End of bit stream reached.\n");
         FT_Close(usbHandle);
         return true;
      }

      usbStatus = FT_Write(usbHandle,datum,6,&written);
      if(written != 6){
         printf("ERROR! Only %u of 6 bytes written successfully.  Aborting...\n",
written);
         break;
      }
      if(usbStatus != FT_OK){
         printf("ERROR writing data!  Code %u.  Aborting...\n", usbStatus);
         break;
      }
      totalWritten += written;
      printf("   %u notes written (%u bytes)\n", totalWritten/6, totalWritten);

//    scanf_s("%s", &test, 100);
   }
   FT_Close(usbHandle);
   return false;
}

bool ConvertChtFile(FILE* fileCHT ,FILE* fileBIT){
   char input[100];
   if(fgets(input,100,fileCHT) == NULL){
      printf("ERROR!!!  Could not read first line header.\n");
      return false;
   }
   if(fgets(input,100,fileCHT) == NULL){
      printf("ERROR!!!  Could not read second line header.\n");
      return false;
   }

   NoteData note;
   while(!feof(fileCHT)){
      if(note.LoadFromCht(fileCHT)){
         if(!note.WriteBinary(fileBIT)) break;  //returns false if EOF reached
         note.Display();
      }else{
         printf("Aborting Conversion...\n");
         return false;
      }
   }
   printf("Convert CHT to BIT Succeeded!\n");
   return true;
}

bool ConvertMidiFile(char* filenameMIDI, FILE* fileCHT){
   rogus::Filedata midiReader(filenameMIDI,NULL);
   int division = midiReader.getDivision();
   if(division < 0){ //SMPTE!!
```

```
      printf("WARNING!!  MIDI timing is SMPTE, no tempo sync!\n");
      division = -division;   //make it positive (actually in ticks/frame...)
   }  //else it's just the number of ticks per beat
   printf("%u ticks per beat\n",division);

   NoteCollection collection;
   for(int track = 0; track < midiReader.getNumTracks(); track++){      //for every
track
      rogus::MidiMsgArrayPtr msgPointer = midiReader.getTrack(track);      //snag
the track's data
      rogus::MidiMsgArray arr = *msgPointer;                        //get rid of the
stupid UtilPtr bs
      printf("Starting Track %u:\n", track);
      NoteData *note;
      for(int msg = 0; msg < msgPointer->length; msg++){                //and go
through the track message by message
         note = new NoteData(arr[msg],track,division);
         if(note->reachedEOF)    //code for if it is a regular MIDI event
            collection.Add(note);   //then add it to the the queue
         else delete note;    //else KILL IT
      }
      printf("Done.\n", track);
   }
   NoteData** allData = collection.Compress();        //unrolls all the events into
one big monster
   sort(allData,allData+collection.length,NoteData::Compare);  //sorts it backwards
   fprintf(fileCHT,"BEAT\tTICK\tINST\tPTCH\tENBL\n");
   fprintf(fileCHT,"4095\t127 \t127 \t127 \t1\n");
   for(int i = collection.length-1; i >= 0; i--)   //reads it back backwards
      allData[i]->WriteCht(fileCHT);
   delete [] allData;                         //free the memory
   printf("SUCCESS!!  CHT file written from MIDI!\n");
   fclose(fileCHT);                        //close the file now because as soon as
FileReader goes out of scope we crash
   return true;
}
```

Appendix B: NoteData.h

```
#pragma once
#include "stdafx.h"
#include "MidiMsg.h"

#define TICKS_PER_BEAT 128

//encapsulates the idea of a note event
class NoteData
{
public:
   NoteData();
   NoteData(rogus::MidiMsg msg, int track, int div);
   bool LoadFromCht(FILE* file);
   unsigned char* ToBinary(); //MUST delete [] the return argument
   bool WriteBinary(FILE* file);
   bool WriteCht(FILE* file);
   void Display();
   NoteData* child;
   unsigned int   beat;
   unsigned char  tick;
   bool        reachedEOF;

   static bool Compare(NoteData* x, NoteData* y);  //used by qsort

private:

   unsigned char  inst;
   unsigned int   ptch;
   bool        enbl;
};
```

Appendix B: NoteData.cpp

```
#include "NoteData.h"

NoteData::NoteData(){
   child = NULL;
}

NoteData::NoteData(rogus::MidiMsg msg, int track, int div)
{
   reachedEOF = true;
   child = NULL;  //ESSENTIAL

   beat = msg.time / div;
   double fractional = double(msg.time % div) / double(div);
   tick = (fractional * (double)TICKS_PER_BEAT);
   inst = track;
   ptch = msg.key();
   if(msg.is_note_end())   //if the message is meant to turn off a note (both by
ending it and setting volume to zero)
      enbl = 0;
   else if(msg.msgt() == rogus::MidiMsg.note_on)
      enbl = 1;
   else
      reachedEOF = false;  //otherwise this is a bullshit message and we don't want
anything to do with it

   if(beat > 4095) printf("ERROR @ %u:%u - Beat is %u > 4095\n",beat,tick,beat);
   if(tick > 127)  printf("ERROR @ %u:%u - Tick is %u > 127 \n",beat,tick,tick);
   if(inst > 127)  printf("ERROR @ %u:%u - Inst is %u > 127 \n",beat,tick,inst);
   if(ptch > 127)  printf("ERROR @ %u:%u - Ptch is %u > 127 \n",beat,tick,ptch);
// if(enbl > 1)    printf("ERROR @ %u:%u - Enbl is %u > 1   \n",beat,tick,enbl);

   if(beat < 0)   printf("ERROR @ %u:%u - Beat is %u < 0\n",beat,tick,beat);
   if(tick < 0)   printf("ERROR @ %u:%u - Tick is %u < 0\n",beat,tick,tick);
   if(inst < 0)   printf("ERROR @ %u:%u - Inst is %u < 0\n",beat,tick,inst);
   if(ptch < 0)   printf("ERROR @ %u:%u - Ptch is %u < 0\n",beat,tick,ptch);
// if(enbl < 0)    printf("ERROR @ %u:%u - Enbl is %u < 0\n",beat,tick,enbl);
}

bool NoteData::LoadFromCht(FILE* file)
{
   unsigned int numLoaded = fscanf_s(file,"%u %u %u %u %u",
&beat,&tick,&inst,&ptch,&enbl);
   if(numLoaded < 5){
      printf("ERROR @ %u:%u - Loaded only %u of 5 fields!\n",beat,tick,numLoaded);
      return false;
   }
   if(numLoaded > 5){
      reachedEOF = true;
      return true;
   }
   reachedEOF = false;
   if(beat > 4095) printf("ERROR @ %u:%u - Beat is %u > 4095\n",beat,tick,beat);
   if(tick > 127)  printf("ERROR @ %u:%u - Tick is %u > 127 \n",beat,tick,tick);
   if(inst > 127)  printf("ERROR @ %u:%u - Inst is %u > 127 \n",beat,tick,inst);
   if(ptch > 127)  printf("ERROR @ %u:%u - Ptch is %u > 127 \n",beat,tick,ptch);
// if(enbl > 1)    printf("ERROR @ %u:%u - Enbl is %u > 1   \n",beat,tick,enbl);
```

```
   if(beat < 0)   printf("ERROR @ %u:%u - Beat is %u < 0\n",beat,tick,beat);
   if(tick < 0)   printf("ERROR @ %u:%u - Tick is %u < 0\n",beat,tick,tick);
   if(inst < 0)   printf("ERROR @ %u:%u - Inst is %u < 0\n",beat,tick,inst);
   if(ptch < 0)   printf("ERROR @ %u:%u - Ptch is %u < 0\n",beat,tick,ptch);
// if(enbl < 0)    printf("ERROR @ %u:%u - Enbl is %u < 0\n",beat,tick,enbl);

   if(beat > 4095 || tick > 127 || inst > 127 || ptch > 127 || beat < 0 || tick < 0
|| inst < 0 || ptch < 0)
      return false;
   //else
   return true;
}

bool NoteData::WriteBinary(FILE* file){
   if(!reachedEOF){
      unsigned char* data = ToBinary();
      fwrite(data,1,6,file);
      delete [] data;
      return true;
   }//else{
   printf("Reached EOF!\n");
   return false;
}

bool NoteData::WriteCht(FILE* file){
   fprintf(file,"%u\t\t%u\t\t%u\t\t%u\t\t%u \n",beat,tick,inst,ptch,enbl);
   return true;
}

unsigned char* NoteData::ToBinary()
{
   //beat - 12
   //tick - 7
   //inst - 7
   //ptch - 7
   //enbl - 1
   unsigned char* data = new unsigned char[6];

   data[0] = (unsigned char) (beat >> 4);                              //so this lops
off the bottom 4 bits of beat, leaving us with the top 8
   data[1] = (unsigned char) (beat << 4) - ((beat >> 4) << 8) + (tick >> 3);
   //bottom 4 of beat + the top 4 of tick
   data[2] = (unsigned char) (tick << 5) - ((tick >> 3) << 8) + (inst >> 2);
   //bottom 3 of tick + the top 5 of inst
   data[3] = (unsigned char) (inst << 6) - ((inst >> 2) << 8) + (ptch >> 1);
   //bottom 2 of inst + the top 6 of ptch
   data[4] = (unsigned char) (ptch << 7) - ((ptch >> 1) << 8) + (enbl << 6);
   //bottom 1 of ptch + enable + trash
   data[5] = (unsigned char)  0;                                    //trash to fill
up the last address

   return data;
}

void NoteData::Display()
{
   printf("beat:%4u tick:%3u inst:%3u pitch:%3u enable:%1u |
",beat,tick,inst,ptch,enbl);

   unsigned char* data = ToBinary();
   printf("char:%3u %3u %3u %3u %3u
%3u\n",data[0],data[1],data[2],data[3],data[4],data[5]);
   delete [] data;
}

//static
bool NoteData::Compare(NoteData* x, NoteData* y){
// NoteData* x = (NoteData*) a;  //cast it back to real life
// NoteData* y = (NoteData*) b;
   if(x->beat > y->beat)
      return true;
   else if(x->beat == y->beat){
      if(x->tick > y->tick)
         return true;
      else
         return false;
   }//else
   return false;

}
```

Appendix B: NoteCollection.h

```
#pragma once

#include "NoteData.h"

class NoteCollection
{
public:
   NoteCollection();
   ~NoteCollection();

   void Add(NoteData* d);
   NoteData** Compress();

   int length;

private:
   NoteData* rootData;
};
```

Appendix B: NoteCollection.cpp

```
NoteCollection::NoteCollection(void)
{
   rootData = NULL;
   length = 0;
}

NoteCollection::~NoteCollection(void)
{
   //outside folks deal with deleting our stuff
}
```

```
void NoteCollection::Add(NoteData *d){
   NoteData* nxt;
   if(rootData == NULL)
      rootData = d;
   else if(rootData->child == NULL)
      rootData->child = d;
   else{
      nxt = rootData->child;
      while(nxt->child != NULL)
         nxt = nxt->child;
      nxt->child = d;
   }
   length++;
}

NoteData** NoteCollection::Compress(){
   NoteData** arr = new NoteData*[length];
   NoteData* nxt = rootData;
   for(int i = 0; i < length; i++){
      arr[i] = nxt;
      nxt = nxt->child;
   }
   return arr;
}
```