Perfect Pitch Sheet Music Maker

John O’Brien, Adam McCaughan

Final Project - 6.111

December 14, 2005

TABLE OF CONTENTS

Introduction 5

Overview of the Audio Handling System 5
Divider/One-32nd Generator 6
Audio Sample Buffer 6
Discrete Fourier Transform 6
Debugging 8
Tone Converter 9
Debugging 10
Tone LUT 11
Debugging 11
Score Converter 12
Debugging 13
Problem abstraction 14
Notation Conventions: Display Specification 14
Pitch 14
Sharps, Flats and Naturals 14
Clefs and Ledger Lines 15
Duration 15
Dotted Notes 15
Rests 15
"p" and "d" Type Notes 16
Beamed Notes 16
Bars and Time Signature 16
Display Architecture 17
Implementation Strategy 19
Frame Buffer and Buffer Manager 19
Design 19
.... 20
Implementation and Debugging 20
Minor Artist Module 21
.... 22
Design 22
Implementation and Debugging 23
Artist Module Major 24
Design 25
Implementation and Debugging 26
Graphics Controller 27
Design 27
Implementation and Debugging 29
Integration 30
Evaluation 30
Conclusions 31

Appendix: Verilog 32
Divider 32
One-32nd Generator 32
DFT 32
Tone Converter 34
ToneLUT 36
Score Converter 37
Frame Buffer 39
Artist Module Minor 42
Artist Module Major 47
Graphics Controller 53
Labkit.v 67

Figure 1: Timing Diagram for the LogicCore FFT 8
Figure 2: Diagram of the Score Converter FSM........ 12
Figure 1: Using Accidentals to Indicate Pitch 14
Figure 2: Using a Key Signature to Indicate Pitch 14
Figure 3: Clef Symbols 15
Figure 4: Note Duration Symbols 15
Figure 5: Common Rest Symbols 16
Figure 6: A "d" note (left) and a "p" note (right) 16
Figure 7: Beamed Notes 16
Figure 8: 4/4 Time Signature 17
Figure 9: Slice Dimensions and Examples 17
Figure 10: Display Layout 18
Figure 11: Frame Buffer Block Diagram 20
Figure 12: Artist Module Minor 22
Figure 13: The "White Line" Bug. 23
Figure 14: Minor Artist Module/Frame Buffer Interface 24
Figure 15: Block Diagram showing state transitions for Artist Module Major........ 26
Figure 16: Meaning of n0, n1, n2 28
Figure 17: Graphics Controller Block Diagram 28

Introduction

The purpose of this project is to allow music to be transcribed automatically. It consists of a microphone, input switches and a video display. The device analyzes music played into the microphone and updates the notes that have been played in real time on the display. The display shows the notes in standard sheet music format, readable by anyone with a basic knowledge of music. The switches are used to control the parameters of the device, specifically the minimum volume necessary to register as a "note," and a harmonic selection device that allows the device take, as input, a range of instruments with varying harmonics. Indeed, the device was shown to work as specified, and properly transcribed music from a range of synthesized notes from a electronic keyboard.

Overview of the Audio Handling System

The audio handling system as a whole functions extremely linearly. At the start of any given millisecond, the DFT module reads its necessary data from the audio sample buffer, and then proceeds directly into computation. As soon as computation of the transform is complete, the outputs head directly and only to the tone converter module, which in turn outputs a characteristic frequency to match against a ROM of predefined frequencies in the tone LUT module. After finding a closest match, the tone LUT sends its corresponding note and octave along to the score converter, whose final function is to determine if the rest of the system should or should not be alerted to the completion of a new note.

Divider/One-32nd Generator

The divider and one-32nd modules serve only to take the input from the 65 MHz clock and output a synchronous signal to signify the passing of milliseconds and 32nd's of a beat. Embedded with their own *count* registers, the modules increments *count* each clock edge until either *count* reaches 65 thousand (1 millisecond) or 1,015,625 (1/64th of a second/120 beats per minute). In the former case, the *one_khz_enable* output pulses to high for one clock cycle, while in the latter, *one_32nd* pulses. In both cases, *count* is reset to zero and its incrementing continues as before.

Audio Sample Buffer

The purpose of the audio sample buffer is to record the 20-bit samples that arrive at 48 KHz from the ac97 and provide them to the FFT as needed. Since the ac97 sets *audio_ready* to high when it has a new sample available, the buffer waits for this change and then records the incoming sample. The address in the memory that it is recorded to is kept track of via the *adptr* register, so any given sample that comes in from the ac97 is written to a specific address, and then *adptr* is incremented so that the following sample will be written to the next address. Due to the fact that the memory is only as many samples as the FFT requires as input in conjunction with the fact that the audio samples need to be taken from a much longer time span than the FFT allows as input, means that the buffer takes the form of a ring buffer. As a ring buffer, once the pointer *adptr* reaches the last writeable address, it loops back and instructs the next audio sample to be written to the beginning of the memory. Since the FFT requires exactly as many samples as are stored, this means no space is wasted holding unused or old samples. Since the times at which the ac97 will provide a new sample are not mutually exclusive with the times that the FFT may need to read data from the memory, the ring buffer is implemented as a dual-port block RAM, with one port serving only to write new samples, and the other to read them as necessary. This addition doubles the size of memory necessary to store the samples, but eliminates the need to deal with troublesome read/write interference.

Discrete Fourier Transform

The discrete Fourier transform module performs a machine-efficient computation of the discrete Fourier transform from the samples stored in the audio sample buffer. At the core of the module is the LogicCore-implemented FFT submodule which, given the correct inputs, performs the actual computation of the transform. At the beginning of every millisecond, signaled by the input *khz_enable*, the DFT module activates its load enable signal, *le*, begins the loading phase of the overall process. Since the audio memory's most current address is passed to the module, but there is no guarantee if it will remain constant through the entirety of the transform, the address *adptr* is recorded immediately at the beginning of the millisecond and held in the register *held_adptr*.

Once the load enable has been asserted, the FFT module begins its acquisition of the audio samples. In order to accomplish this, it needs to be able to know which

memory address it should read as its real input, *xn_re*—the imaginary input *xn_im* is tied to zero because audio signals are purely real signals. Using the register *held_adptr* as its base, the module adds on *xn_index*, an output from the LogicCore FFT that represents the index of the input data requested, and puts this summed value into the register *readaddr*. This register reads is output from the DFT module into the audio sample buffer, where after one clock cycle, the appropriate is output from the memory and read directly into the FFT input, *xn_re*. Once the FFT submodule is finished loading the final sample, it proceeds directly into computing the transform, signaling this fact by setting the internal *busy* signal to a logical high. During this time, no special considerations are given to the read address outputs or sample inputs; controlling these is unnecessary because they only affect the FFT during its loading phase.

During the computation of the transform, the module remains static. However, once the *busy* signal drops—signaling the completion of the computation—the DFT module sends the LogicCore FFT submodule a single-cycle high pulse to its input *fft_unload,* and so begins its unloading phase after a 7-cycle delay that changes the output order of the unloaded data from "bit-reversed" into "natural," which is both conducive to debugging and easier for the tone converter module to work with. In the unloading phase, the FFT submodule produces three outputs of relevance: *xk_index*, the index k of the N-point discrete Fourier transform; *xk_re*, the signed real value of the kth point of the transform; and *xk_im*, the signed imaginary value of the kth point of the transform. At each clock cycle that the clock enable, *ce*, is high, the FFT submodule returns an incremented value of *xk_index*, along with its associated real and imaginary values, which the DFT module then outputs into the tone converter module that follows it.

The DFT module tracks the completion of the unloading phase with the 1-bit *unload_done* register, which simply casts itself as a logical high when *xk_index* reaches the final (N-1) point of its output. Though the DFT is capable of completing its full three phases long before a new millisecond arrives, no early warning arrives for the next *khz_enable,* so the choice was made to reset the FFT submodule and have it ready for new input immediately following *unload_done*. In order to prepare the submodule for the new loading phase, the *unload_done* is propagated, at each clock enable, linearly through *sclr*, *nfft_we,* and *fwd_inv_we* [Figure 1].

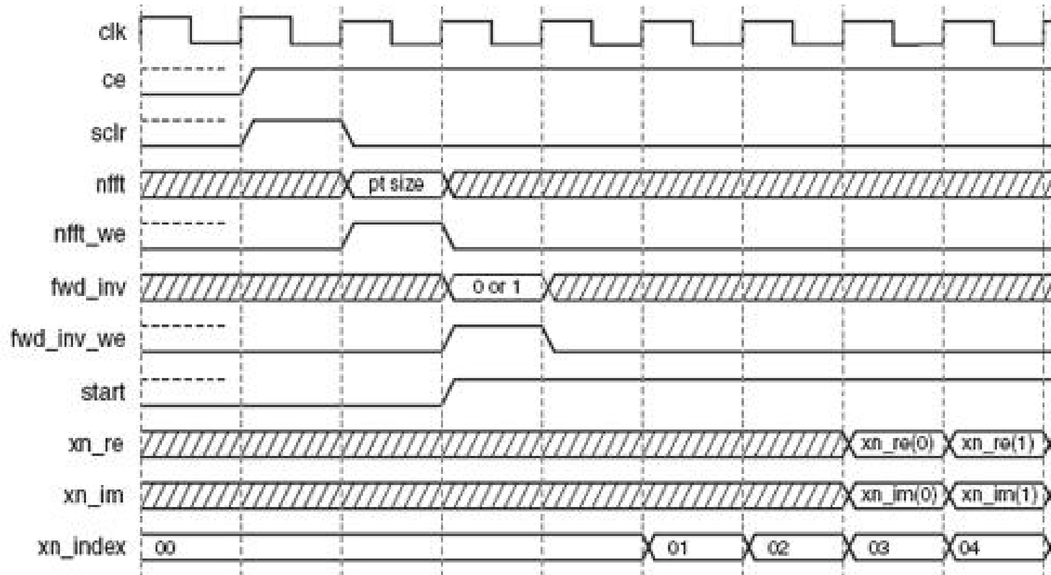


Figure 1: Timing Diagram for the LogicCore FFT

The first signal, *sclr*, performs a synchronous reset, while the second and third allow specification to occur for the input-length and transform-direction respectively. The specification for *nfft_we* occurs because, when it goes high, the submodule sets how many points it is going to input and compute according to the signal *nfft*, which is tied to the constant parameter fft_length. Similarly with *fwd_inv_we*, its corresponding value *fwd_inv* is set to a logical '1,' which tells the FFT submodule that it will be computing the forward transform, and not the inverse transform. This "reset-and-set" procedure allows the DFT module to respond properly and immediately as soon as the next millisecond arrives.

Debugging

The DFT module was by far the hardest of all the modules to debug. In order to have easy access, realizable access to the data, the code from the vga_graph2_buf.v module had to be adapted away from taking as input from the FPGA's analog-to-digital converter, and instead take its data, data clock, and data start from the FFT. Once, however, that was accomplished, simple button controls allowed between viewing the audio signal going into the DFT, and viewing the output from the FFT. It was after the implementation of this side-module that the cause of the incorrect frequency the tone converter module was reporting was discovered—initially, no option to examine the fundamental frequency was available, and it was discovered that the dominant frequency for the electric piano being used was in fact the third harmonic. Also, extremely useful to test the DFT system was the side-module ddstester.v. With the help of the LogicCore Direct Digital Synthesizer, producing a specified sine wave to appear at the input port of the DFT became as simple as raising and lowering a frequency register that connected to one of the input ports of the DDS.

Tone Converter

Relying primarily on frequency-domain input from the discrete fourier transform, the tone converter module takes the unsigned versions of the real and imaginary components of the DFT's output and intelligently decides what the characteristic frequency should be. Since the computation of the characteristic frequency requires the DFT to have finished computing the transform, the tone converter module waits until it receives *fft_unload* to begin producing any meaningful output. Once it receives *fft_unload*, the module clears the registers *max_magsqsum*, *max_indexsum*, *fund_index, fund_found, fund_freq,* and *dom_freq*, all of which are necessary to tracking the characteristic frequency. Then, 7 cycles later, the unsigned versions of the DFT outputs *xk_re, xk_im,* and *xk_index* appear at the inputs of the module in the form of *re_pipe, im_pipe,* and *index_pipe* respectively. From there, the three inputs are pipelined into *re, im,* and *index* respectively, in order to reduce total combinational delay time from the output of the DFT to the computation of the magnitude.

As *index* increments each cycle, and along with it new values of *re* and *im* appear, a magnitude for the fourier output is calculated by individually squaring the eight most significant bits of the real and imaginary inputs and adding them together. The resulting magnitude, *magsq*, is kept track of for three total iterations of the logic/clock cycles, by loading *magsq* into the register *magsq2*, and *magsq2* into the register *magsq3* at every positive edge of the clock. If, however, the index is less than two, *magsq2* and *magsq3* are loaded with values of zero, the reason being that the inputs previous to index zero are invalid and that the input associated with index zero itself (the DC offset of the DFT's input signal) is meaningless to the function of the tone converter module. The value of *magsq2*, the "center" magnitude of the three inputs looked at in any given cycle, is summed with half the value of each of *magsq* and *magsq3* and stored into the register *magsqsum*. Since this register is the only one used in the logic for finding the characteristic frequency, it effectively means that the computed "magnitude" is actually the shaped sum of three points of data in the frequency domain.

In order to give the user as wide a variety of instruments as possible to be able to use on the microphone input, the algorithm used to find the characteristic frequency *char_freq* must be adaptable. With adaptability in mind, the design choice was made to allow the user to select between reporting either the microphone input's fundamental frequency (the lowest harmonic), or its dominant frequency (the harmonic with the most energy). The user is able to select between reporting the fundamental and dominant frequencies by setting the FPGA input *switch[5],* which is taken as input into the tone converter via *fund_select*. A value of 1 reported to *fund_select* selects the dominant frequency, while a value of 0 selects the fundamental frequency. To extend the adaptability even further, the user is allowed to input the desired *mag_threshold* for which the frequency domain magnitude must exceed in order to be reported. If no part of the frequency spectrum achieves the desired threshold, the characteristic frequency is set to zero and, farther down the line of modules, the score converter module will not report it as a note. In the same manner as *fund_select*, these values are dynamically assigned based on the FPGA inputs *switch[4:0]*, which appear as inputs to the tone converter module as the input *mag_threshold*. The ability to set the magnitude threshold is especially useful with a microphone that is in the presence of external noise. Since the design requires that a certain volume be reached, in noisy environments the threshold

may be raised in order to keep from reporting false frequencies and, in turn, allowing false notes to appear.

So, with three registered magnitudes of the input signal summed and *index* above the value of two and increasing at a regular frequency, the search for the fundamental and dominant frequencies begin. For the fundamental frequency, the shaped magnitude *magsqsum* is incremented linearly along each point of the frequency until a value is found that exceeds the input value of *mag_threshold.* Once found, a logical 1 is stored into the register *fund_found* and, simultaneously, the relevant frequency point, *index-1* is stored into the register *fund_index.* Once *fund_found* is set to one, the index of the fundamental frequency is no longer allowed to be updated because the first harmonic has been found. In the dominant frequency, however, this value is stored (because the threshold is the same for both frequencies), but may be updated if the shaped magnitude *magsqsum* finds another, larger value as it is updated with *index*. The search for both frequencies only stops once *index* becomes greater than half the *fft_length*, because the nature of our purely-real signal is to create a completely symmetric discrete Fourier transform across this border. The last half of the transform, then, provides no new information and would only serve to incorrectly update the dominant frequency.

The actual frequencies, in hertz, of the dominant and fundamental are calculated by multiplying the values of their respective indices by 48000 (sample rate of the ac97) and dividing the result by *fft_length*, which gives the whole system a maximum reportable frequency of 24000 Hz—well above the highest frequency perceivable by the human ear and far above the pitch necessary to meet the design specification. The actual output of the characteristic frequency, *char_freq*, is then output based on the user's selection between outputting the dominant or fundamental frequency. The characteristic frequency is only accessed by the tone LUT module once it is done parsing the spectrum—a single-cycle *tc_done* high pulse is sent upon reaching the halfway point of the spectrum.

Debugging

The main method of debugging for the Tone Converter was the liberal use of the hex display output. Since the both the index and output of the DFT could readily be relied upon to show up on the VGA monitor, it was decided to use the hex display so that both, heavily interconnected, modules could have rational outputs examined at the same time. By outputting *max_indexsum, max_magsqsum*, and *char_freq* along with watching the DFT output on the monitor, it was possible to immediately determine whether or not the tone converter was, for instance, reporting the correct index, whether the calculation of the frequency in Hz was correct, and what the minimum magnitude necessary as an input to be able to pick up notes from a particular instrument. One major problem came along in the form of the FFT reporting extraordinarily high values for the zeroeth index of the system. This was solved by realizing that it was DC offset of the signal that was skewing things, and this problem was resolved by implementing the (index > 2) lines seen in the final Verilog code.

Tone LUT

The primary function of the tone lookup-table module is to receive the characteristic frequency, *char_freq*, from the tone converter module, compare its value with a ROM of predefined notes, and report the note and octave that the frequency represents. Two ROMs, *prefreqrom* and *noteoctave*, are core to its functionality, the former of which cycles through a list of predefined frequencies associated with valid notes, and the latter of which holds the predefined frequencies' relative note and octave. The ROMs hold values corresponding to each of the 12 notes in the scale, from octaves 0 through 8, giving each a total 108 locations. The predefined frequencies ROM contains rounded values of the frequencies corresponding to each note/octave combination, spanning a range of 30 Hz to 13 Khz. The width of each location, then, for *prefreqrom* is 15 bits. The second ROM, *noteoctave,* requires a width of 8 bits because the first four bits are used to represent the numeric range of notes (A as 0x0, A# as 0x1 … G# as 0xC) and the last four bits represent the associated octave.

Upon reception of its initializing signal, *tc_done*, from the tone converter, the tone LUT module knows it has a valid, constant *char_freq* input. When it receives *tc_done*, the module resets its internal address register, *addr*, to zero, and begins incrementing through the various predefined frequencies. At each cycle, the wire *abs_diff_r_c* combinationally evaluates the absolute difference between a new predefined frequency output from the *prefreqrom* and the characteristic frequency it receives as input from the tone converter module. Should this value be smaller than the current smallest-difference found (computed by taking the absolute value of *char_freq* and subtracting the register *best_freq*), this new, closest, value will be stored into the *best_freq* register, and its location in the rom, *addr_pipe2* will simultaneously be stored into the *best_freq_addr* register. The *noteoctave* ROM's input address is tied to *best_freq_addr*, and so as the module cycles through all the possible values of *addr* and updates its closest-fitting frequency, *noteoctave* always makes the closest-fitting note and octave available. Finally, in the cycle at which the ROM read address *addr* has reaches 108, the module output *note* is registered to the top four bits of the note and octave ROM's output, *note_octave*, the module output *octave* is registered to the bottom four bits of *note_octave*, and the ROM read address is latched until the characteristic frequency is determined again in the next millisecond. The exception, however, to the outputs of *note* and *octave* is if the characteristic frequency is reported to be zero, in which case both 4-bit outputs are registered as the reserved value 0xF in order to signify that a rest is occurring.

Debugging

Similarly to the tone converter module, the Tone LUT debugging was best served by a hex output. One of the primary problems that the module came across was the offset of its *best_freq_addr* to the address that truly was the best frequency. By outputting *octave*, *note*, *rom_freq,* and *best_freq* as a precautionary measure, the problem was soon realized—incorrect pipelining—because at a given stopping point in the module's runtime, the *rom_freq* displayed would be offset by a value of two.

Score Converter

As the final step in handing off information about a note that was played to the video portion of the entire Perfect Pitch system, the score converter module's primary responsibility is the tracking of what note and octave were played, what absolute time they started at, and what duration of note it represents in standard sheet-music format [Figure 2]. Due to the design of the timing in the overall system [Figure XX], the score converter module can rely on having a valid *note* and *octave* input available to it at the beginning of every khz_enable and for some lengthy period after that. The 32nd of a note timing, conversely, holds no promise about the validity of *note* and *octave*, and so once score converter receives the *one_32nd* pulse from the one32ndgen module, it must latch *one_32nd* into *new_32nd* until the beginning of the next millisecond to begin its function towards output.

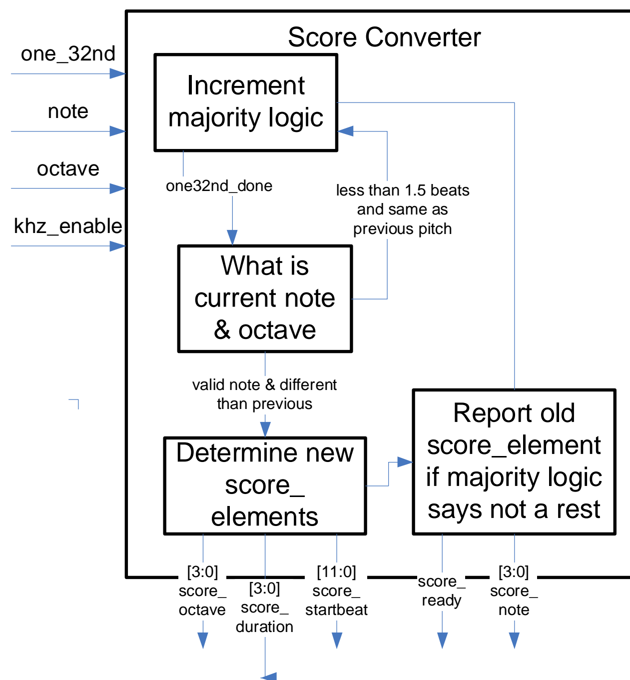


Figure 2: Diagram of the Score Converter FSM

When *khz_enable* signals the beginning of score converter's function and *new_32nd* is asserted, the module has the wire *same_pitch* combinationally check to see if current inputs *note* and *octave* are the same as the values *old_note* and *old_octave* recorded at the previous 32nd beat. If *same_pitch* holds true, and the total duration elapsed, *duration_count*, with this same pitch is less than 48 32nds (the longest note available is a beat and a half), the module only increments the *duration_count* register and waits until the next *khz_enable* to perform additional logic. If, instead, the current pitch is different than the previously recorded one, or *duration_count* shows that the note has reached the longest displayable value, this signals that it's time to update the outputs *score_octave, score_startbeat,* and *score_duration*, and to assert *note_finish* to begin the next step of reporting the output of the *score_* registers.

One of the features central to this module is its ability to do determine whether or not a note should be conveyed, using majority logic. At the beginning of every millisecond, score converter looks at the value of its input *note* and check to see it is a value from 0 to 11—a valid note. If indeed a valid note, it increments that particular note's location in the dual-port BRAM *note_count*; if not, it increments the register *silence*, to show that the last millisecond elapsed was a rest. When the score converter decides it is time to fire off information about a note by asserting *note_finish*, it first steps through each location in *note_count,* storing the maximum value contained within in into the *maj_max* register and its associated note into *maj_note*—this register holds the value of the note with the greatest number of milliseconds associated during the period the same pitch was playing.

Having thus determined *score_octave, score_startbeat,* and *score_duration*, the final step for the score converter module is to decide whether to or not to register *score_note* as the note held in *maj_note* or to mark it down as a rest and not to report any of the new *score_* values. It decides whether or not to ignore the note based on the ratio between *maj_max* and *silence*. Unless the note with the greatest number of milliseconds associated with it is four times greater than *silence*, the new *score_* values are not reported. If it is greater than four times silence, *score_ready* sends a pulse and the new *score_* elements are available at the output. The reason for this skewed ratio is in order to prevent false reports of notes during the period where the sound from an instrument may be just rising or falling off: during these times the magnitude threshold may make the tone converter waver between reporting a frequency and reporting zero every millisecond. By ensuring a large majority of "note reports" during the milliseconds that pass, the *silence* produced by wavering quickly overpowers *maj_max* and allows for a sharp cutoff.

Debugging

Surprisingly, the score converter was relatively easy to debug—yet like the two modules that linearly precede it, it was prudent to provide meaningful hex data. For this purpose, a ticker was created in the form of *hex_sc*. Dividing the outputs into quarters, *hex_sc* simultaneously was able to track the changes of *score_octave*, *score_note, score_startbeat* and to count the increments of *score_ready*. A particular problem that arose occurred in the output of *score_ready*: it would pulse once at the beginning of the note, and once at the end of the note. Given that it is only supposed to pulse when a note has been completed, this led to some very strange output on the video portion of the Perfect Pitch. By closely watching *score_octave* and *score_note*, it was found that the *score_note* was not being properly reset to its value of 0xF after the pulse of *score_ready.* This led to an immediate discharge of another *score_ready* the next time a note was played, because it thought it was changing over from one valid note to the other, and should pulse, when in fact it should have been coming from a rest to a valid note, in which case it would not pulse.

Problem abstraction

The display side of the project was responsible for taking information about new notes (pitch, octave, duration, start beat) and adding them to a graphical representation of the piece played so far. Standard music notation convention was decided as the most easily accessible graphical representation.

Notation Conventions: Display Specification

Pitch

Standard musical notation evolved from methods of notating vocal music (specifically plainchant), which means that it was primarily intended to display pitch, with the duration of the notes implied by the rhythm of the song’s lyrics. It encodes information about the pitch of a note in its vertical position on a five line staff.
This meant that the display was required to adjust the positioning of a symbol on a staff depending on its pitch. The standard method for measuring pitch is the twelve note chromatic scale. The twelve note scale includes the pitches A to G#. However as far as the positioning of a note is concerned A and A# can be regarded as identical, so the display needed to position notes based on an 8 note scale.

Sharps, Flats and Naturals

Sharps and flats are one semitone above or below their corresponding natural tone. They can be notated using a key signature and accidentals. A key signature is a frame at the beginning of the notation that specifies which tones are to be played as flats, sharps and naturals. Notes will be interpreted by this scheme unless a different pitch is indicated using an accidental. An accidental is a small symbol placed next to the note’s symbol indicating that it should be played sharp or flat.

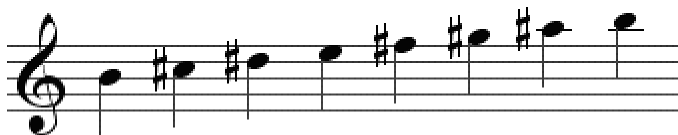


Figure 3: Using Accidentals to Indicate Pitch

Figure 4: Using a Key Signature to Indicate Pitch

A key signature reduces clutter in the notation, however it was decided to use only accidentals in the display. This decision was made because a key signature would have required either user input or a clever analysis of the relative frequency of sharp, flat and natural notes of a given pitch. Neither of these approaches would have added much to the usability of the project.

Not using a key signature also meant that all semitones could be represented by the sharp symbol (since A# = Bflat etc.) These decisions required the display to add a sharp symbol to notes of the appropriate pitch.

Clefs and Ledger Lines

The position of a note on the staff shows its pitch relative to other notes on the same staff. Its absolute pitch is shown relative to the clef. The clef sets a line on the stave to a particular note and octave. There are a large number of clefs, their symbols are shown in Figure 5.

Figure 5: Clef Symbols

Using clefs reduces the need to use ledger lines: small lines added to notes whose pitch is above or below the staff. It was decided to make the display capable of supporting the four most commonly used clefs (treble, bass, alto and tenor) and introduce ledger lines if time permitted. This meant that the display was required to take a user input of the desired clef and alter the positioning of the notes on the stave accordingly.

Duration

Duration is encoded symbolically. The standard symbols used are shown in Figure 6. To simplify the project only the symbols 1 to 1/32 were used.

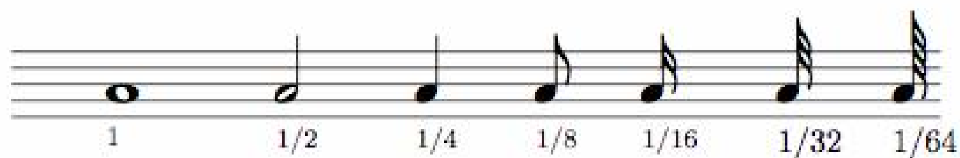


Figure 6: Note Duration Symbols

Dotted Notes

To reflect the fact that musical notes are not tied to the durations shown in Figure 6 music notation allows the duration of a note to be extended by 150% through the addition of a dot. It is also possible to extend the note by 175% using two dots, 187.5% for three dots etc., but this usage is rare. The display was required to add dots to notes of the appropriate duration.

Rests

Rests have a corresponding set of symbols representing duration. Since they do not represent a pitch they should always be placed on the same line of the staff. The display was required to infer from the durations and start beats of incoming notes whether a rest had occurred between them and draw the appropriate sprite.

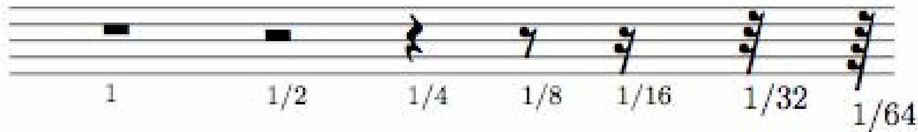


Figure 7: Common Rest Symbols

"p" and "d" Type Notes

By convention, notes are displayed differently depending on whether they are on/above the central line of the staff or below it. For high notes the stem and flag are put below the note head, for low notes they are put above. In this report and in the verilog code associated with it notes with the stem below are referred to as "p notes" and those with the stem below are referred to as "d notes". The resemblances to a "p" and a "d" can be seen in Figure 8.

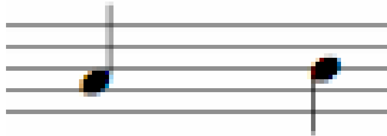


Figure 8: A "d" note (left) and a "p" note (right)

The display needed to be able to tell which category a particular note fell into and select the appropriate sprite.

Beamed Notes

Another musical convention is to "beam" runs of short notes (notes equal to or shorter than 1/8). The purpose of this convention is to make scores easier to read. Sometimes notes of differing duration are joined as well, as shown in Figure 9.

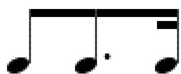


Figure 9: Beamed Notes

Often short notes that follow a linear increase in pitch are also beamed, however to simplify the display it was decided that only consecutive notes of the same pitch and basic duration (not counting dots) would shown as beamed.

Bars and Time Signature

The time signature gives the meter of a particular musical piece, specifically how many beats there are to a bar. It is shown at the start of a piece of music, near the clef (). The bars represent the periodic "pulse" of the music. To show these elements the display needed to take a user input of the time signature, display it at the start of the piece and insert bar lines based on its value. The display was specified to deal with 4/4, 2/2, 4/2, ¾ and 6/8 (the most common time signatures).

Figure 10: 4/4 Time Signature

Display Architecture

To give a high resolution and allow notes to be displayed sharply XVGA was used (1024x768 pixels, 60Hz). This required that the project be clocked to 65MHz. To be able to display notes of a reasonable pitch above and below the staff, the height of a single "slice" was set to 14 note heads tall. The width of a slice was chosen to be 32 pixels. This value has the advantage of being a factor of 1024 (screen width). A reasonable aspect ratio sets the height of a slice to 140 pixels (i.e. 10 pixels per note head). The resulting slice dimensions are shown in Figure 11. These dimensions allowed 4 rows of 30 slices each to be displayed on the screen with reasonable margins, as shown in Figure 12.

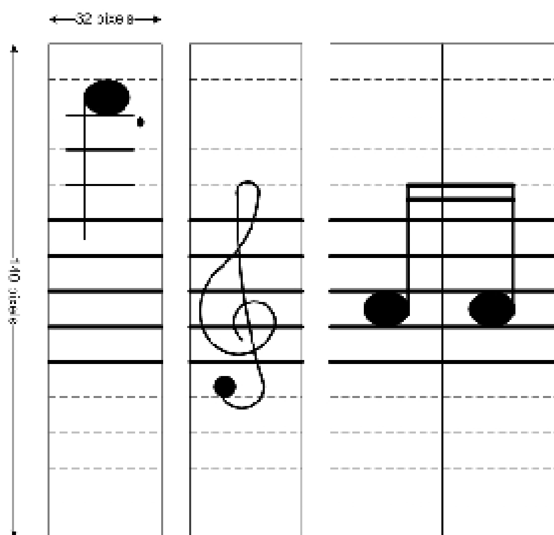


Figure 11: Slice Dimensions and Examples

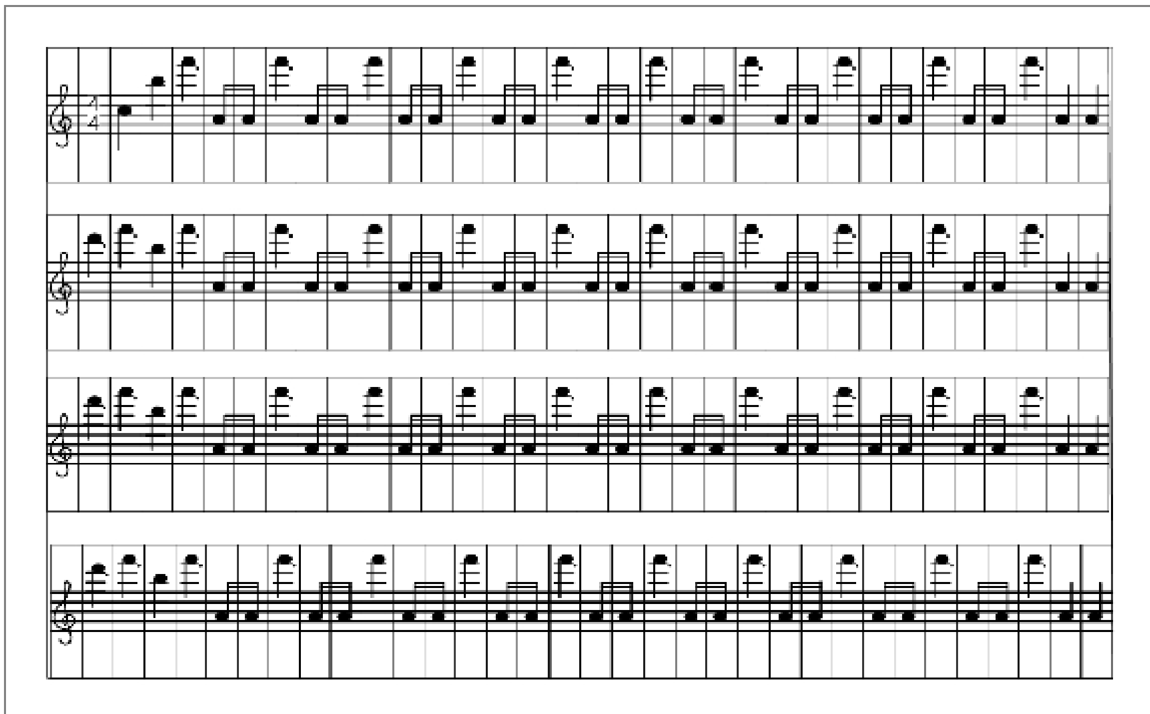


Figure 12: Display Layout

Given the complexity of the display, recalculating the value for every pixel once a frame (as in Lab 4: Pong Game) appeared unfeasible. Instead a frame buffer was planned. This would allow the display modules to alter the screen incrementally. To make communication with the frame buffer straightforward it was decided to make the data word length for the frame buffer 32 bits. This would allow slices to be sent to the frame buffer at the rate of one row per clock cycle.
The convention of beamed notes meant that a new note could alter how the previous two notes were displayed. This in turn meant that immediately after receiving a new note from the audio module the display modules would have to calculate new pixel values for three slices – the new note and the two preceding slices. This was broken down into three units:

Graphics controller Takes in a pitch, octave, duration, start beat and ready signals from the audio module. On a ready signal it will select the appropriate sprites to be drawn (taking into account p/d notes, beamed notes etc.) for the last three slices and passes them along with pitch and octave to the artist module major.
Artist module major On a ready signal from the graphics controller passes the information about the three sprites in series to the artist module minor. It also calculates the starting x and y position of each sprite to be drawn. The x position is passed directly to the frame buffer while the y position is passed to the artist module minor. This is because the artist module minor will need to increment the y position each time it draws a new row.
Artist module minor This will take information about which sprite to draw, its pitch/octave and a starting y position from artist module major. It will load the sprite

from a sprite memory, mix it with a staff and pass the correct pixels and y coordinate values to the frame buffer.

Originally it was planned to have two memories for graphics elements: one for sprites and one for graphics elements that always have the same position on the staff (i.e. clef, time signature, rests). There would have been separate loading routines for both memory types, plus another routine that would draw a blank frame. Beamed notes were intended to be drawn using ¼ note sprites and then adding beams between them with combinational logic. However as implementation progressed it became clear that all these functions could be smoothly implemented using a single sprite memory.
Sprite dimensions were chosen to be 32x80 pixels (treble clefs take up about 7 note heads = 70 pixels). Since both d and p type note symbols were included in these memory slots, and because they are about 60 pixels tall, the sprites have their note heads at different positions. To accommodate this the first row of each sprite includes the y coordinate of the note's center relative to the top of the sprite in binary.

The labkit provides 4MB of ZBT SRAM and 2.6Mbit of BRAM (144x18kbit). The main memory requirement of the project will be the frame buffer. To provide a resolution of 1024x768 pixels with 3 bits per pixel it will need a capacity of 2.4Mbit. This can easily be accommodated in a ZBT SRAM. If the refresh rate is 60Hz then the ZBT will need to provide a new 3 bit pixel value at the rising edge of the 65 Mhz pixel clock. The ZBT can output 36 bits every clock cycle, and can be clocked at up to 167 Mhz so it should be able to meet these demands.

Implementation Strategy

The implementation strategy was to work backwards from the frame buffer to the graphics controller. This would allow each consecutive module to be tested on the display with some simple input faking modules. Initially modules would be implemented without any capability to produce anything other than the basic functions of the display (i.e. no beams, rests, sharps, dots). The rationale behind this decision was that it is much easier to add features to a working project. Once the bare bones of the project were completed and all the display modules had been successfully tested and integrated extra features could be added.
The modules are described in the order they were constructed.

Frame Buffer and Buffer Manager

The frame buffer holds a black and white image of the whole screen in memory. It must output all of these pixel values to the display in series every frame. It must also take pixel values and coordinates from the artist module and update the frame buffer image to take into account the changes.

Design

Originally the frame buffer was intended to use a ZBT memory as the memory element for the frame buffer. This would have had two advantages: firstly the zero turnaround between writing and reading would maximize the number of pixels that could be saved

into the buffer every frame, secondly putting the frame buffer in a ZBT would free up BRAMs for the FFT and the sprite memory.
Since each slice is only written into the frame buffer once any lost data during writes will show up as glitchy pixels on the display. The periodic transition from writing to reading must be coordinated such that no data is lost. To allow this a busy signal is generated as the frame buffer switches from writing to reading. This busy signal pauses the artist module so that no data is lost.
The buffer manager must generate an address for each row of 32 pixels based on their x and y coordinate so that they can be saved in an unique location in memory. This is generated using the formula below. Slice_x has been divided by 32 because that is the width of a slice.

address = {slice_y[9:0], slice_x[9:5]};

This generates a unique memory address for every possible slice coordinate within the 1024x768 display area. However slice_x[10:0] and slice_y[9:0] can take values outside this area. Although this situation should never occur intentionally the frame buffer is designed to disable the memory write when the address becomes invalid. This feature is exploited in other modules by setting slice_y to 768 whenever it is desired to avoid writing to the frame buffer.
Generating an address to retrieve pixel information based on hcount and vcount is more complicated because of the blanking regions.

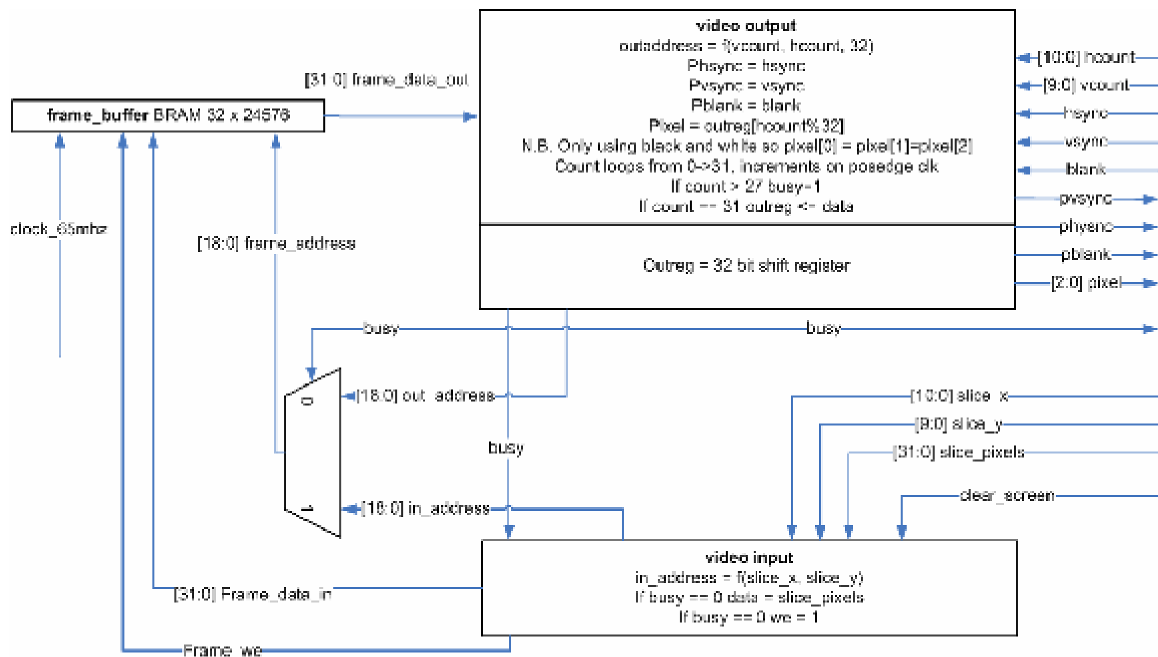


Figure 13: Frame Buffer Block Diagram

Implementation and Debugging

The code supporting the ZBT was not available when work began on the frame buffer, so a BRAM was used as a temporary measure. This meant increasing the number of busy cycles per frame. Since the BRAM performed acceptably and there was no shortage of memory blocks it remained in place after the ZBT became available.

A basic module to fake a flashing block input to the frame buffer was built to check its function. The module was designed such that the user could move the flashing block around the screen using the up, down, left and right buttons.
The input faking module showed that the buffer did not display the flashing block when it was moved to x=0. Additionally when the block was moved to x=288 the flashing block was displayed in two locations. This implied that there was a problem with reading from the memory since it was not possible for one data sample to be written to two memory locations. Examining the code showed that while the read address was being dealt with correctly during the blanking period between frames, it was not correctly assigned in the blanking period between lines. Altering the logic for *out_address* fixed the problem.
Another bug was found when frame buffer was tested with the artist module.

Minor Artist Module

The minor artist module takes as inputs the number of the sprite to be drawn along with its pitch and octave and generates a stream of 32 bit wide pixel values with corresponding y coordinates that are passed to the frame buffer. It interfaces with the major artist module using the *artist_start* and *artist_done* signals. When new sprite data (*n_sprite, n_octave, n_pitch)* is available the major artist module generates a one clock cycle high pulse on *artist_start.* This signal activates the drawing process. Once the minor artist module finished drawing it responds to the major artist module with a one clock cycle pulse on *artist_done.*

The minor artist module also receives a *clef* signal from the user. It needs this because the offset of a sprite on the staff is a function of its pitch, octave and the clef being used for the piece.
When the minor artist module receives a *busy* signal from the frame buffer it must stop sending out new information until the busy signal is deasserted.

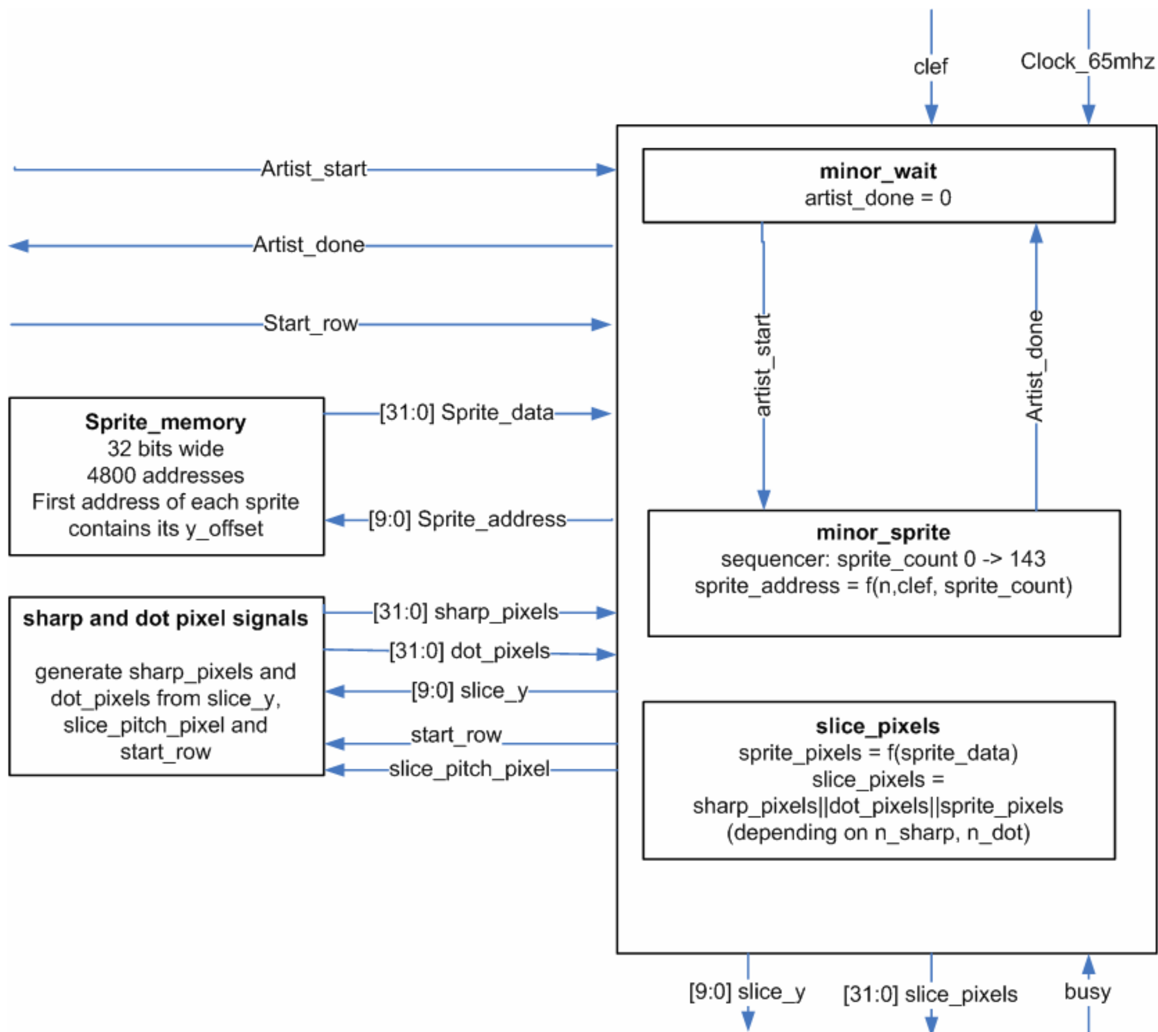


Figure 14: Artist Module Minor

Design

The module is based around a sequencer that, on receiving an *artist*_start signal, increments the variable *sprite_count* from 0 to 143 before generating an *artist_done* pulse and returning to a wait state. In the first few clock cycles the module determines the offset of the sprite from the top of the slice being drawn. It does this by loading the first line of the sprite, which contains the offset of the note head from the top of the sprite. The offset of the line the note is to be drawn on from the top of the slice is calculated in parallel using the *clef, n_pitch* and *n_octave* signals. The address in the sprite memory that matches up with the first line of the slice, *initial_address*, is then calculated.
Once the *initial_address* has been found the address is incremented with *sprite_count* to make *current_address*. The assertion of the memory address lags one cycle behind receiving the data from the *sprite_memory* BRAM.
The data from the BRAM is used to generate the 32 bit *sprite_pixels* signal. First the line of the slice being drawn is checked to see whether it is 60, 70, 80, 90 or 100 pixels from the top of the slice. These values correspond to the staff lines, so those rows are set to black lines. (*current_address* - 1) is then assessed to see whether the address that

generated the data currently coming from the BRAM lies within the 80 pixel range of the sprite being drawn. If it is not and that row of *sprite_pixels* has not been detected as a staff line then *sprite_pixels* is made to be white. If (*current_address* - 1) is within the valid range *sprite_pixels* is set to the data signal coming from the BRAM.
Two other 32 bit wide pixel signals are generated. *Sharp_pixels* is set to contain the sharp symbol whenever the row being drawn is in the right range. Likewise *dot_pixels* contains the information to make a note dotted.
Dot_pixels, sharp_pixels and *sprite_pixels* are then combined. Depending on the values of *n_dot* and *n_sprite* the correct signals are put through a bitwise OR gate to give *slice_pixels,* the output to the frame buffer. *Slice_y* is a simple function of the sequencer variable, *sprite_count,* and the initial y coordinate of the slice, *start_row.*

Implementation and Debugging

To test the module's performance a simple two sprite ROM was prepared by editing a .coe file. Sprite number 0 was given a cross image, while sprite 1 was given a diamond. A simple module was prepared to give the minor artist module and the frame buffer a series of sprite and coordinate instructions.
The input faking module allowed a couple of minor bugs to be fixed quickly and also showed up a more serious one. The sprites contained white lines and repeated pixels, as shown in Figure 15. Since the errors occurred 32 rows apart it seemed likely that the fault was related to the busy signal.

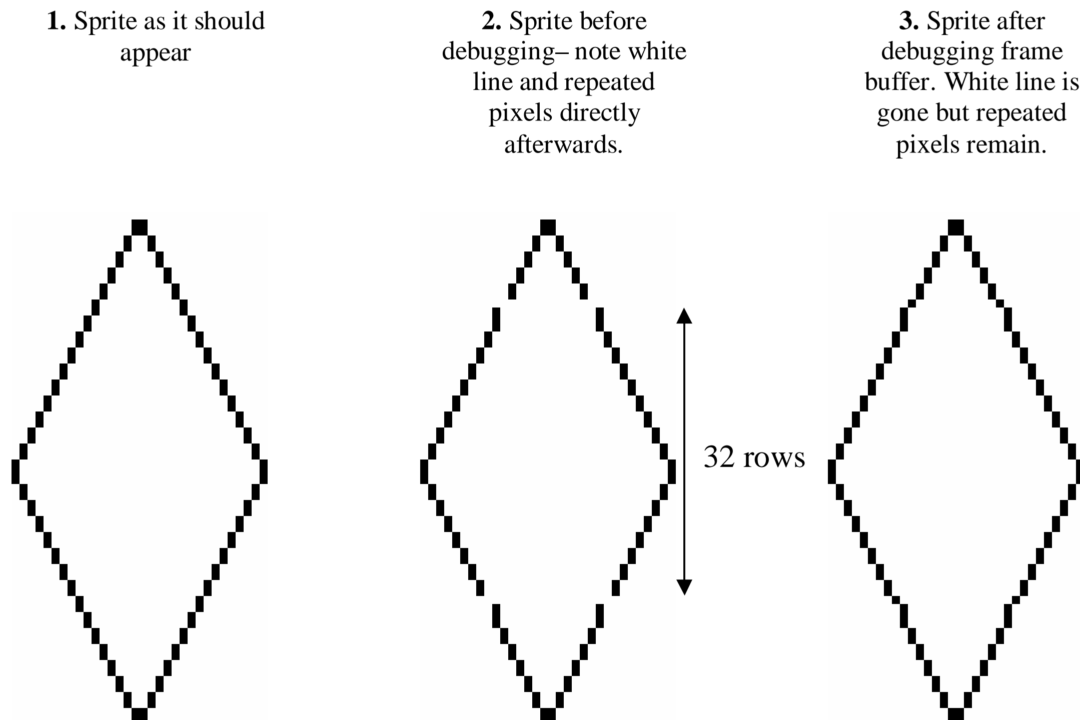


Figure 15: The "White Line" Bug.

Examining the interface between the frame buffer and the minor artist module revealed that the white line was caused by inappropriate usage of synchronous logic. Frame_address and frame_data_in were generated from slice_y and slice_pixels respectively within an "always

@(posedge clock)" block. This meant that they were written to the buffer a cycle after they were received. However as can be seen in

Figure 16 when busy goes high the delay causes a row of data not to be written to the frame buffer (c, [c]). This omission causes the white line seen in Figure 15. It can also be seen from the timing diagram that after the busy cycle "d" is written to both "[d]" and "[e]". This mismatch causes the repeated pixel bug and was due to the logic that handled busy signals within the artist module. On receiving a busy signal the module did not increment *sprite_count* until *busy* was deasserted. However, since the address for reading from *sprite_memory* was prepared one clock cycle before the corresponding data could be used as an output the pause caused the *slice_pixels* and *slice_y* signals to get out of sync.

The white line bug was solved by changing the calculation of *frame_address* and *frame_data_in* to combinational logic. The repeated pixel bug was solved by changing how the *sprite_address* signal was assigned during a *busy* signal.

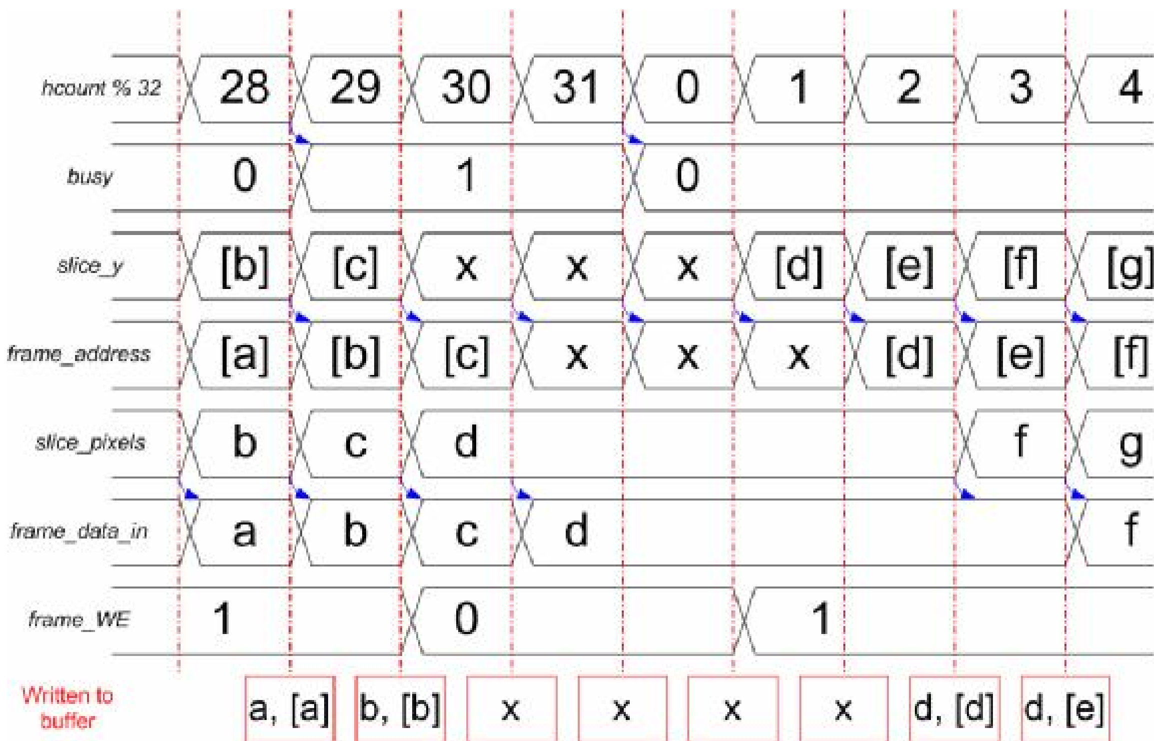


Figure 16: Minor Artist Module/Frame Buffer Interface

Red lines represent posedge clock. "[b]" represents the address to which the word of data, "b", should be written to. Data is written to the frame buffer memory based on the values of *frame_address* and *frame_data_in* whenever a positive clock edge occurs and *frame_WE* is high. It can be seen that the clock cycle delay between the inputs *slice_y* and *slice_pixels* and the data and address signals for the frame buffer memory cause the word of data "c" to be lost. It can also be seen that after the busy signal finishes there is a temporary mismatch in *frame_address* and *frame_data_in* that causes data word "d" to be written to address "[e]".

Artist Module Major

Artist module major takes in inputs describing slices to be drawn and their relative position and feeds the information to artist module minor one at a time, governing

communication using the *artist_start* and *artist_done* signals. It also calculates the starting x and y coordinates of each slice (*slice_x* and *start_row*).

Design

The module is based around a four state FSM. The FSM starts in the *wait* state and remains there until it receives a *graphics_ready* signal from the graphics controller module. It then passes through the states *draw_n2*, *draw_n1* and *draw_n0*. At each state transition it shits the outputs *n_sprite, n_octave, n_pixel, n_sharp, n_dot, slice_x* and *start_row* to reflect the slice being drawn and then sends an *artist_start* pulse. The signals prefixed with "n" are merely reassigned from *n2_sprite, n0_dot* etc., but *slice*_x and *start*_row must be calculated from the current slice's *countslice.* This signal tells artist module major how many slices have been drawn to the screen before the current one. Artist module major must take into account margins and the number of slices per row to convert this signal into the correct values of *slice_x* and *start_row.* When it receives an *artist_done* pulse it moves on to the next state, and when it returns to the *wait* state it sends a *display_ready* signal to graphics controller informing it that it is ready to receive a new set of slices.

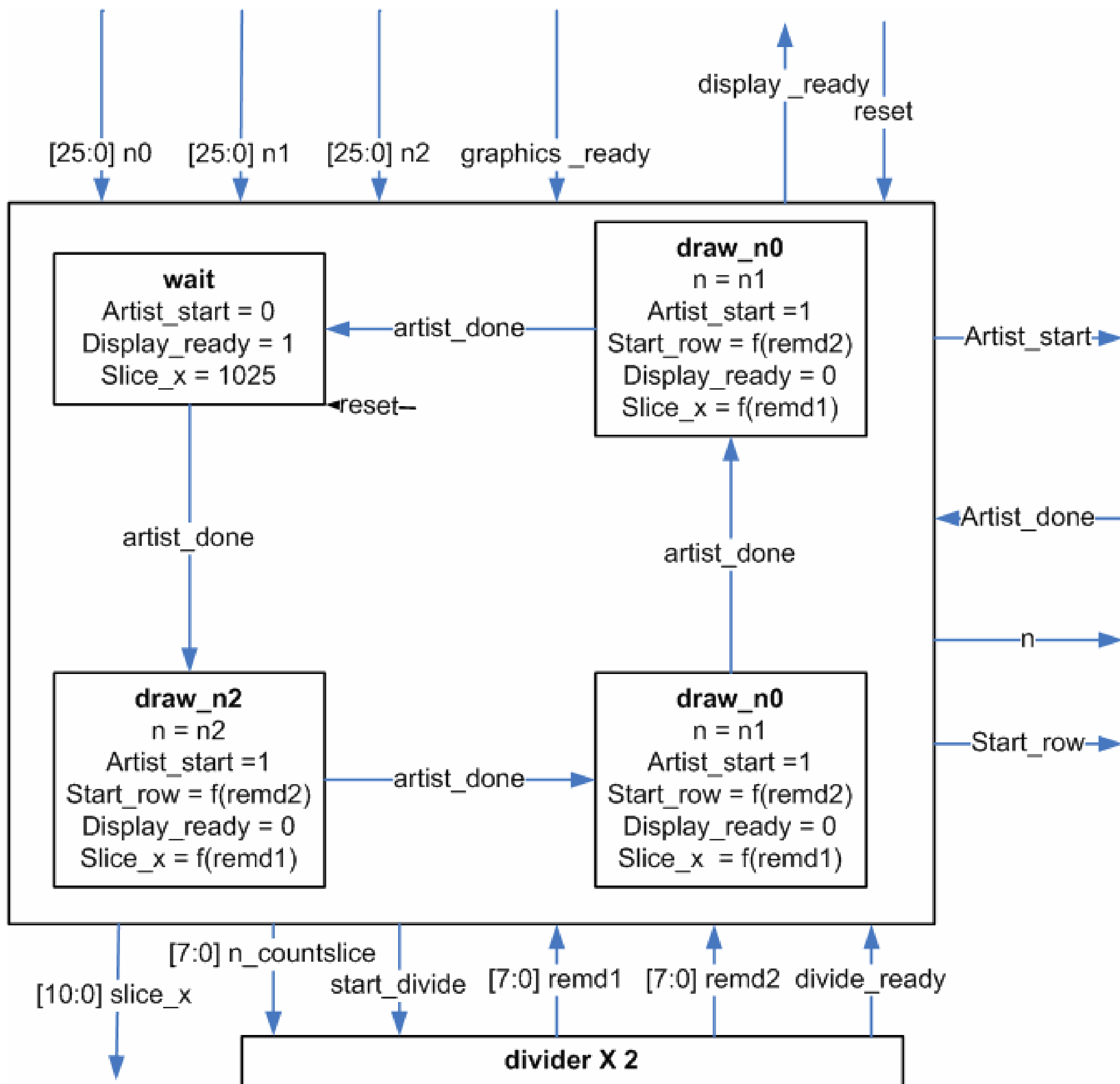


Figure 17: Block Diagram showing state transitions for Artist Module Major
The signals *n2_sprite, n2_countslice* etc. have been collapsed into *n2, n1, n0* and *n* to make the diagram clearer. There are two instances of the divider module, one for calculating *start_row* and one for *slice_x*.

Implementation and Debugging

The calculation used to determine *slice_x* is shown below.

```
slice_x = countslice/slices_per_row * pixels_per_slice + x_margin;
```

Previously all divisions had been by powers of two, so it was surprising when this line of code failed to synthesize. The design had naively assumed that the compiler would implement a combinational divider for a divide sign, but this was not the case. Therefore synchronous dividers were added to the module as shown in Figure 17. Initially the Xilinx IPCoregen dividers were used, but these proved unsuitable because they could not be simulated in the version of ISE used. Also, these dividers were heavily pipelined to

maximize throughput, which was an inefficient use of resources for an application were only latency was important.
Instead a simple divider was adapted from code available at http://www.ece.lsu.edu/ee3755/2002/l07.html. The code was edited to make the start and ready signals reliable and to reduce the bit width of the dividend and divisor. The only changes made to the main FSM were to activate *start_divide* instead of *artist_start* on a state transition and to trigger *artist_start* from *dividers_ready*.
A less elegant solution was used to deal with a random bug that caused the major FSM to occasionally hang in one of the states *n2_draw, n1_draw* or *n0_draw*. No reason could be found for this bug, so an inelegant but effective solution was implemented. When entering a new state a sequencer, *cludge_count,* was triggered. This would count to 800 clock cycles and then cause the FSM to move to the next state. If an *artist_done* signal were received before this point the FSM would change state as usual. The delay caused by 800 clock cycles is ample time for a slice to be drawn by artist module minor and not long enough to give any noticeable lag to the display. With hindsight this bug may have been due to long combinational delays within artist module minor. Since these combinational delays have since been greatly reduced it may be that this fix is no longer necessary.

Graphics Controller

The graphics controller converts information about a new note event into high level instructions for updating the display. These instructions are passed onto the artist module which performs the low level manipulation of pixels. The graphics controller deals with the context dependent issues of choosing between a "p" and a "d" note and of adding beams to short notes of the same duration and length.

Design

The block diagram for the graphics controller is shown in Figure 19. The module was designed with implementing an extra playback function in mind so some of the states used are superfluous (i.e. controller_save was intended to save new score_elements, controller_pmaster would have kept track of where pages of notes begin and end in the score_memory). n0, n1 and n2 refer to the last three slices to be output to the artist modules. For instance in Figure 18 if the display had just finished drawing a new note, during the drawing process n0, n1 and n2 would have referred to the notes shown.

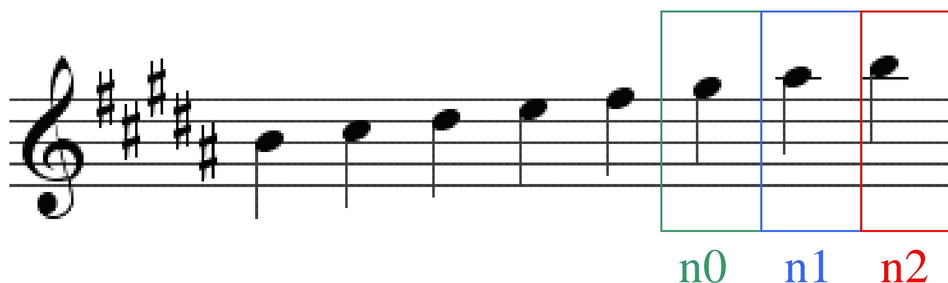
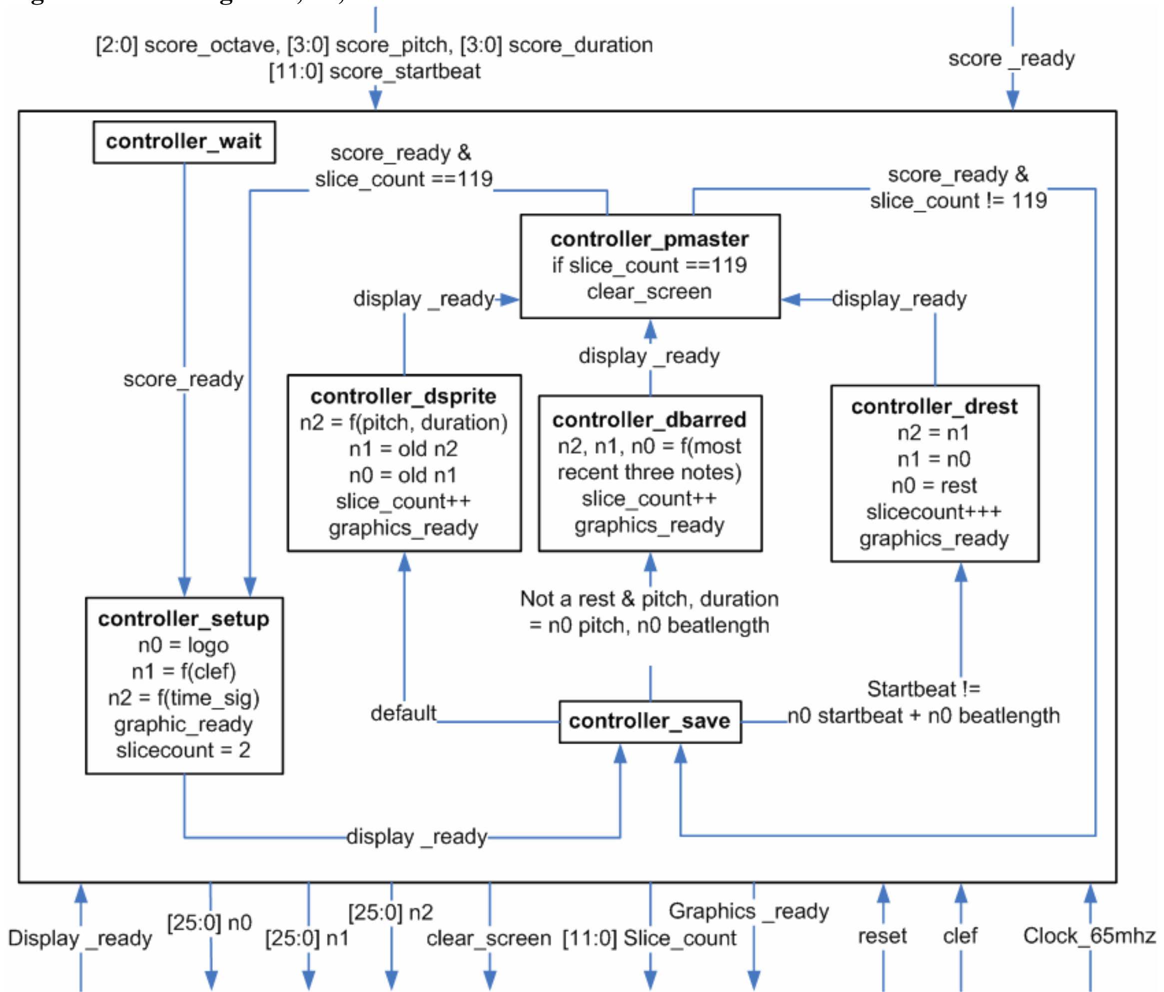


Figure 18: Meaning of n0, n1, n2



+

Figure 19: Graphics Controller Block Diagram

The design is based around a sequencer, *slice_count.* This keeps track of how many slices have been drawn to the screen. It is passed to the artist modules along with the sprite and pitch information to allow slices to be properly positioned on the screen.

The system begins in the *controller_wait* state and returns there on a reset. On receiving a *score_ready* signal from the audio part of the project it moves to the setup state. It sets the sprites to display a “perfect pitch” logo, the appropriate clef and a time signature and sends a *graphics_ready* signal. All of these graphical objects should appear in the same place on the staff regardless of the clef chosen, but artist module minor is hardwired to interpret sprites’ positions relative to the current clef. Therefore the pitch and octave sent along with these “static” objects has to be changed according to the clef selected. The signals *static_octave* and *static_pitch* accomplish this task and are generated by a small look up table.

Once artist module major signals that the logo, clef and time signature have been drawn with the *display_ready* signal the FSM increments slice_count to three and switches to the *controller_dsprite* state. This state only draws one new slice and leaves the previous two unchanged. The controller decides on the sprite to be drawn (“p” or “d”) by

comparing the note's pitch and octave to the pitch and octave of the middle line of the staff under the current clef. Once this new slice has been drawn by the artist modules the FSM checks to see if the end of the screen has been reached and then waits for a new score_ready signal.

Drawing beamed notes and rests follows a similar procedure. The signals *go_bar* and *rest* are assigned combinationally to be high when the conditions are met to warrant a beamed note or a rest respectively (note it the two are mutually exclusive). If a beamed note is to be drawn it needs to be known whether the previous three notes are also beamed. Two logic signals, *bar1* and *bar2* are generated for this purpose. If the old value of n0 has equal duration and pitch to old n1 then *bar1* goes high. If the old value of n2 is beamed with the old value of n1 then *bar2* goes high Depending on the values of *bar1* and *bar2* slices n0 -> n2 will be assigned either a left, middle or right hand beam sprite or the same value as the slice that used to be in front of them before slice_count was incremented.

The FSM enters the *controller_drest* state when a pause has elapsed between the end of the last note and the start of the new note. The module determines this by comparing the startbeat of the new note with the startbeat of the old note plus its duration in beats.

For rests the graphics controller needs to assign two new sprites – the rest and the new note played. This means that *slice_count* is incremented by two. Since the difference will not always be exactly 1/16, ¼ etc. the module approximates to the nearest value rest sprite.

Implementation and Debugging

Although the graphics controller worked reliably when it was integrated with the artist modules and the frame buffer the display was very "noisy". The background flickered and there were many individual pixels that were displaying the wrong values. Some notes were displayed several pitches below what they should have been at first and then were redrawn correctly when the next note was drawn in.

It was theorized that this poor performance was due to timing errors caused by the relatively high clock frequency. Examining the synthesis report revealed that there was a maximum combinational delay of 24ns along one of the data paths in the display module. The clock period was only 15ns (65MHz), so it seemed likely that this was what was causing the glitches. The delay appeared to be mainly occurring in the artist module minor, which contains a lot of combinational logic to compute the positions of the sprites on the screen. Firstly the outputs of each module were pipelined, then when this had little effect the internal logic of the artist module minor was pipelined. The data paths were drawn out by hand to see where pipelines registers could be placed and five possibilities were found. Two of these pipelines stages were implemented, bringing the delay down to around 15.5ns (the other pipelines lay outside the critical step). It was difficult to see how a third useful pipeline stage could be fitted into the module, but by examining the details of the synthesis report it was seen that a single element of combinational logic was taking almost 5ns to complete. The exact nature of this element was found using the rtl schematic tool. The element in question dealt with the verilog statement below.

```
Assign slice_pitch_pixel =
      140 - (n_octave_p1*7 + n_pitch_p1 - slice_bottom_p1)*5
;
```

The rtl schematic showed that this statement was being implemented as two multipliers in series: one to perform (x*7) and another to perform the (y*5) operation. This approach did not seem optimal, so the verilog code was changed to make it clear that all multiplication could be performed in parallel.

```
assign slice_pitch_pixel =
      140 - (n_octave*35 + 5*n_pitch - 5*slice_bottom);
```

The rtl schematic showed that this changed the layout to do the multiplications in parallel. This took approximately 3ns off the delay for this assign statement without changing its result and made the critical combinational delay path drop to 12.54ns. When the pipelined code was synthesized and loaded onto the FPGA the display glitches had almost all been fixed. The only glitch remaining was an occasional, seemingly random pitch shift in a new note that was repaired when the next note was drawn. Since this seemed less frequent after pipelining than beforehand it is speculated that this bug is also dependent on too long a combinational delay (although the simulated delay path is below the critical value of 15ns it is still close and when implemented the combinational delay could exceed the clock period).

Integration

The integration of the two halves of the project was very straightforward. The point at which work was divided had been carefully chosen to give as simple an interface as possible, and that meant that getting the two modules to operate together only took about two hours. There were, however, some integration issues when changes were made to the project. The display remained very sensitive to long combinational delay paths and when the width of the FFT was expanded beyond a certain value it became very glitchy, displaying artifacts similar to those seen before the artist module minor was pipelined. Also, adding the code for dealing with rests to the integrated project caused the display to become very glitchy, even though the rest code worked without display issues when the display half of the project was driven by a user input.

Evaluation

For the display side of the project improvements are needed to futher reduce combinational delay paths and remove the remaining, seemingly timing related, glitch. This would also allow the FFT width to be expanded, improving the accuracy of the audio segment, and the verilog code for handling rests to be inserted.
One suggestion would be to reduce the level of computation by increasing the memory used per sprite. In the current implementation of the project the sprite memory slots are not large enough to allow both "p" and "d" type notes to be drawn with their note heads the same distance from the top of the sprite. For this reason the y offset of the note head is encoded in the first row of the sprite. Manipulating this value to give the sprite address that corresponds to the first row of the slice requires several steps of computation which all occur within the current maximum combinational delay path. Increasing the sprite size

to 32x120 pixels would mean that "p" and "d" notes could be displayed with their note heads on the same line and so remove much of the computation required. There were sufficient BRAMs available at the conclusion of our project to accomplish this without compromising other processes that use the memory.

Another possible improvement would be to greatly simplify the logic for writing to the frame buffer. Instead of having a busy signal that halts the rest of the display modules for 3/32 clock cycles the artist module minor could be modified to write each new value of *slice_pixels* and *slice_y* to the frame buffer four times consecutively. This would guarantee that the value would be written on at least one of the four clock cycles (since busy is only 3 cycles long). Any data that were sent to the frame buffer while busy was high would not corrupt existing values because *frame_WE* would be set low.

Extra features which could be implemented on top of the existing project that would improve its functionality would be the ability to play back a synthesized recording of the piece played after recording was over. Support for drawing bars could also be added relatively easily. By increasing the size of the sprite memory the current beamed note functionality could be extended to deal with trains of notes following a steady rise in pitch and to notes of different durations.

Conclusions

Overall, the project was an astounding success. Almost all of the functionality laid forth in the design document was implemented [only rests were excluded, and even they worked on a sidemodule]. Though connecting the two halves of the system proved to be a difficult task, in the end the foresight of making this connection as minimal as possible paid off greatly. Once connected, it became a simple matter of working out the peculiarities of each side's code, and creating fixes for functions that disrupted others—for instance, too large of an FFT module caused major glitching in the display half of the system.

Appendix: Verilog

Divider

```
module divider(clock_65mhz, reset, one_khz_enable);
	input clock_65mhz, reset;
	output one_khz_enable;
	reg [15:0] count;

	always @ (posedge clock_65mhz) begin
		if (count < 65000) count <= (reset ? 0 : count+1);
		else count <= 0;
	end

	assign one_khz_enable = (count == 65000);

endmodule
```

One-32nd Generator

```
module one32ndgen(clock_65mhz, reset, bpm, one_32nd);
	input clock_65mhz, reset;
	input [7:0] bpm;
	output one_32nd;
	reg [23:0] count=0;

	always @ (posedge clock_65mhz) begin
		if (count < 1015625) count <= (reset ? 0 : count+1);
		else count <= 0;
	end


	assign one_32nd = (count == 1015625);

endmodule
```

DFT

```
module dft(clk, ce, khz_enable, adptr, xn_re, le, readaddr,
		fft_unload, xk_index, xk_re, xk_im);


	parameter fft_length = 1024;		// (fft_length-1)
	parameter fft_index_bits = 10;		// (fft_index_bits-1)
	parameter fft_in_bits = 20;		// (fft_in_bits-1)
	parameter fft_out_bits = 31;		// (fft_out_bits-1)

	input clk;
	input ce;
	input khz_enable;
	input [(fft_index_bits-1):0] adptr;
	input [(fft_in_bits-1):0] xn_re;
	output reg le;
	output reg [(fft_index_bits-1):0] readaddr;
```

```
       output reg fft_unload;
       output [(fft_index_bits-1):0] xk_index;
       output [(fft_out_bits-1):0] xk_re;
       output [(fft_out_bits-1):0] xk_im;

   // FFT:  WIRES AND ASSIGNMENTS

       wire [4:0]    nfft = fft_index_bits;
// set NNFT = fft_length (2^fft_index_bits)

       reg           nfft_we;
// Will be set to high after calculation

       wire          fwd_inv = 1;
// Compute forward transform

       reg           fwd_inv_we;
// Will be set to high after calculation

       reg           sclr;

       wire [(fft_in_bits-1):0]   xn_im = 0;
       wire [(fft_index_bits-1):0]      xn_index;
       wire        fft_rfd;
       wire        fft_busy;
       wire        fft_dv;
       wire        fft_edone;
       wire       reset = 0;
       reg        unload_done;

       reg [(fft_index_bits-1):0] held_adptr;

       reg old_fft_busy;


       always @ (posedge clk) begin

                 // captures khz_enable in le until ce occurs
              le <= khz_enable ? 1 : ce ? 0 : le;

              if (ce) begin
                     // grabs the address pointed to in the ring buffer
                     held_adptr <= le ? adptr : held_adptr;
                     // Specifies the read address in the ring buffer
                     readaddr <= xn_index + held_adptr;

                     // xk_index reaches (fft_length-1), unload_done pulses 1
                     unload_done <= (xk_index == (fft_length-1));

                     sclr <= unload_done;
                     // then synchronously reset

                     nfft_we <= sclr;
                     //    then set nfft size

                     fwd_inv_we <= nfft_we;
                     //    then write fwd_inv = 1

                     old_fft_busy <= fft_busy;
                     // Makes fft_unload go high for one cycle

                     fft_unload <= ~fft_busy & old_fft_busy;
              // as soon as busy signal drops from high to low
```

```
            end
        end

    ac97fft myfft(          xn_re,          // real data, input
                            xn_im,          // imaginary data, input(always zero)
                            le,     // start data loading & conversion, aka start
                            fft_unload,     // start unloading data (busy must be
                                            // finished)
                            nfft,           // Tied
                            nfft_we,        // high to rewrite nfft
                            fwd_inv,        // forward or inverse, input
                            fwd_inv_we,     // write enable for fwd_inv, input
                            sclr,
                            ce,
                            clk,            // system synchronous clock
                            xk_re,          // output real data
                            xk_im,          // output imaginary data
                            xn_index,       // index of input data (output)
                            xk_index,       // index of output data (output)
                            fft_rfd,        // ready for data, out
                            fft_busy,       // high while core is computing fft
                            fft_dv,         // data valid, output
                            fft_edone,      // early done strobe, output
                            fft_done);      // fft complete strobe, output

endmodule
```

Tone Converter

```
module toneconv(clk, fund_select, mag_select, re_pipe, im_pipe, index_pipe,
fft_unload, char_freq, tc_done, hex_tc);

        parameter fft_length = 1024;            // (fft_length-1)
        parameter fft_index_bits = 10;          // (fft_index_bits-1)
        parameter fft_in_bits = 20;             // (fft_in_bits-1)
        parameter fft_out_bits = 31;            // (fft_out_bits-1)

        input                           clk;
        input                           fund_select;
        input [4:0]                     mag_select;
        input [(fft_out_bits-1):0]      re_pipe;
        input [(fft_out_bits-1):0]      im_pipe;
        input [(fft_index_bits-1):0]    index_pipe;
        input                           fft_unload;
        output [14:0]                   char_freq;
        output reg                      tc_done;
        output [63:0]                   hex_tc;

        wire [5:0] mag_threshold =      mag_select*2;
        wire [5:0] mag_threshold_fund = mag_select*2;



        reg [(fft_out_bits-1):0]   re;
        reg [(fft_out_bits-1):0]   im;
        reg [(fft_index_bits-1):0] index;

        always @ (posedge clk) begin
```

```
            re <= re_pipe;
            im <= im_pipe;
            index <= index_pipe;
      end



      // Magnitude of the fft
      wire [16:0] magsq =       re[(fft_out_bits-1):(fft_out_bits-8)]
                                *re[(fft_out_bits-1):(fft_out_bits-8)]
                +               im[(fft_out_bits-1):(fft_out_bits-8)]
                                *im[(fft_out_bits-1):(fft_out_bits-8)];

      reg [16:0]   magsq2;
      reg [16:0]   magsq3;
      reg [18:0]   magsqsum;
      reg [18:0]   max_magsqsum;
      reg [18:0]   fund_freq;
      reg [18:0]   dom_freq;
      reg          fund_found;
      reg [(fft_index_bits-1):0] max_indexsum;
      reg [(fft_index_bits-1):0] fund_index;


      always @ (posedge clk) begin

                  tc_done <= index == (fft_length/2);

                  magsq2 <= (index > 2) ? magsq : 0;
                  magsq3 <= (index > 2) ? magsq2 : 0;
                  magsqsum <= (index > 2) ? (magsq/2)+magsq2+(magsq3/2) : 0;

                  // Reset index and magnitude if starting a new unload
                  // Otherwise, compare old values
                  // Only the first half of the DFT is useful
                  if (index <= (fft_length/2)) begin
                        max_magsqsum <= fft_unload ? 0 : (magsqsum >
max_magsqsum) ? magsqsum : max_magsqsum;

                        max_indexsum <= fft_unload ?
                         0 : (magsqsum > max_magsqsum) ?
                         index : max_indexsum;

                        fund_index <=   fft_unload ?
                         0 : ((max_magsqsum > mag_threshold_fund)
                         & ~fund_found) ?
                         index-1 : fund_index;

                        fund_found <=   fft_unload ?
                         0 : (max_magsqsum > mag_threshold_fund);
                  end

                  fund_freq <= fft_unload ?
                   0 : fund_found ? (fund_index * 48000 / fft_length) : 0;
                  dom_freq <= (max_magsqsum > mag_threshold) ?
                   (max_indexsum * 48000 / fft_length) : 0;

      end
            assign char_freq = fund_select ? dom_freq : fund_freq;
            assign hex_tc = {max_indexsum,max_magsqsum,17'b0,char_freq};
endmodule
```

ToneLUT

```
module tonelut(clk,char_freq,tc_done,note,octave,hex_lut);
	input clk;
	input [14:0] char_freq;
	input tc_done;
	output reg [3:0] note;
	output reg [3:0] octave;
	output [63:0] hex_lut;

	reg [6:0] addr;
	reg [6:0] addr_pipe;
	reg [6:0] addr_pipe2;
	reg [6:0] best_freq_addr;
	wire [14:0] rom_freq;
	reg [14:0] best_freq;
	wire [7:0] note_octave;
	reg update_n_o;

	wire [14:0] diff_r_c = rom_freq - char_freq;
	wire [14:0] diff_b_c = best_freq - char_freq;
	wire [14:0] abs_diff_r_c = diff_r_c[14] ? ~diff_r_c[13:0] + 1 : diff_r_c;
	wire [14:0] abs_diff_b_c = diff_b_c[14] ? ~diff_b_c[13:0] + 1 : diff_b_c;

	prefreqrom prefreqrom1(addr, clk, rom_freq[13:0]);
	assign rom_freq[14] = 1'b0;

	noteoctave noteoctave1(best_freq_addr, clk, note_octave);


	always @ (posedge clk) begin

			// Increment addr if less than # of predefined frequences
			if (addr <= 108) addr <= addr + 1;
			// Otherwise, hold addr until tc_done resets it to zero
			else addr <= tc_done ? 0 : addr;

			addr_pipe <= addr;
			addr_pipe2 <= addr_pipe;

			best_freq <= (abs_diff_r_c < abs_diff_b_c) ?
			 rom_freq : best_freq;
			best_freq_addr <= (abs_diff_r_c < abs_diff_b_c) ?
			 addr_pipe2 : best_freq_addr;

	// Update note and octave only once it reaches the end of the list
	// If char_freq is zero (just noise), spit out invalid note/octave (0xF)
			if ((addr == 108) & (addr_pipe == 107)) begin
				note <= (char_freq == 0) ? 4'hF : note_octave[7:4];
				octave <= (char_freq == 0) ? 4'hF : note_octave[3:0];
			end

	end


	assign hex_lut = {octave,12'b0,note,17'b0,char_freq};

endmodule
```

Score Converter

```
module scoreconv(clk,khz_enable,note,octave,one_32nd,
startbeat,score_octave,score_note,score_duration,score_startbeat,
score_ready,hex_sc);

	input clk;
	input khz_enable;
	input [3:0] note;
	input [3:0] octave;
	input one_32nd;
	output reg [11:0] startbeat;
	output reg [3:0] score_octave;
	output reg [3:0] score_note;
	output reg [3:0] score_duration;
	output reg [11:0] score_startbeat;
	output reg score_ready;
	output [63:0] hex_sc;

	reg [3:0]	old_octave;
	reg [3:0]	old_note;
	reg [5:0]	duration_count;
	reg [11:0] beatcount=0;

	reg new_32nd;
	reg note_finish;

	// Updated only on (new_32nd & khz_enable)
	wire same_pitch = (old_octave == octave) & (old_note == note);

	reg old_note_finish;
	reg [10:0] note_count[11:0];
	reg [10:0] maj_max;
	reg [3:0]  maj_note;
	reg [3:0]  n;
	reg [10:0] silence;



	always @ (posedge clk) begin


		// Latch one_32nd until khz_enable comes

		new_32nd <= khz_enable ? 0 : one_32nd ? 1 : new_32nd;
		beatcount <= one_32nd ? beatcount + 1 : beatcount;

		if (~note_finish) begin
			if (khz_enable) begin
			 if (note < 12) note_count[note] <= note_count[note] + 1;
				else silence <= silence + 1;

				// New-32nd block
				if (new_32nd) begin
					old_octave <= octave;
					old_note <= note;

					// Pitch_ready? block
					if (same_pitch & (duration_count < 48))
```

```
                                        duration_count <= duration_count + 1;
// Pitch that's been tallied is ready to be fired
                                  else begin
                                        score_octave <= old_octave;
                                        score_startbeat <= startbeat;

if           (duration_count >= 48)     score_duration <= 11;      // Dotted 1
else if      (duration_count >= 32)     score_duration <= 5;       // 1
else if      (duration_count >= 24)     score_duration <= 10;      // Dotted 1/2
else if      (duration_count >= 16)     score_duration <= 4;       // 1/2
else if      (duration_count >= 12)     score_duration <= 9;       // Dotted 1/4
else if      (duration_count >= 8)      score_duration <= 3;       // 1/4
else if      (duration_count >= 6)      score_duration <= 8;       // Dotted 1/8
else if      (duration_count >= 4)      score_duration <= 2;       // 1/8
else if      (duration_count >= 3)      score_duration <= 7;       // Dotted1/16
else if      (duration_count >= 2)      score_duration <= 1;       // 1/16
else if      (duration_count == 1)      score_duration <= 0;       // 1/32
else   score_duration <= 15; // invalid score_duration for debugging

       // Signal that there's a new note incoming with 1/32nd of time on it
       // and that the previously tallied pitch is ready to be fired
                                        duration_count <= 1;
                                        note_finish <= 1;
                                        startbeat <= beatcount;
                                        maj_max <= 0;
                                        maj_note <= 0;
                                  end

                           end
                    end          // khz enable block
             end                 // note_finish block
             else begin          // Majority Logic begin
                    if (n <= 11) begin
                           maj_max <= (note_count[n] > maj_max) ?
                            note_count[n] : maj_max;

                           maj_note <=  (note_count[n] > maj_max) ?
                            n : maj_note;

                           note_count[n] <= 0;
                           n <= n + 1; end
                    else begin
                           score_note <= (maj_max > silence*4) ?
                            maj_note : 4'hF;
                           note_finish <= 0;
                           n <= 0;
                           silence <= 0;
                    end

             end

             // Determines the clock cycle during which note_finish completes
             old_note_finish <= note_finish;
             score_ready <= ~note_finish & old_note_finish
                                        & (score_note != 4'hF);

       end


//     For debugging

       reg [15:0] count_ready=0;
       always @ (posedge clk) count_ready <= score_ready ?
```

```
        count_ready + 1 : count_ready;
       reg [15:0] count_32nd=0;
       always @ (posedge clk) count_32nd <= one_32nd ?
        count_32nd + 1 : count_32nd;

       assign hex_sc = {count_32nd,count_ready,12'b0,octave,12'b0,note};

endmodule
```

Frame Buffer

```
// buffer_manager.v
// Provides a monochrome frame buffer for a 1024x768 pixel display
// John O'Brien
// 11/20/05

module buffer_manager (        hcount, vcount, hsync, vsync, blank,
                               pvsync, phsync, pblank, pixel,

                               //Debug I/O
                               /*
                               //outreg,
                               frame_address,
                               in_address, out_address,
                               frame_data_in, frame_data_out,
                               frame_we,
                               */

                               slice_x, slice_y, slice_pixels, busy,
                               clear_screen,
                               reset, clock_65mhz
                               );

//XVGA I/O
input [10:0] hcount;           //current pixel x from xvga
input [9:0] vcount;            //current pixel y from xvga
input hsync, vsync;            //sync pulses
input blank;                   //blanking pulse from XVGA
output pvsync, phsync;         //Sync signal outputs
output pblank;                 //Blanking output for XVGA
output [2:0] pixel;            //Pixel output to XVGA

//ARTIST MODULE MINOR I/O
input [10:0] slice_x;          //current slice X coord
input [9:0] slice_y;           //current slice Y cocord
input [31:0] slice_pixels;     //values of current slice

output busy;                   //Busy signal to artist module

//GRAPHICS CONTROLLER I/O
input clear_screen, reset;     //Clear_screen has the same effect as
                               //reset, but only applies to
                               //buffer_manager. Used by graphics
                               //controller to clear the screen when the
                               //current one is full

//GLOBAL I/O
```

```
input clock_65mhz;


//DEBUG I/O (Used for pulling values out to simulate)
/*
//output [31:0]outreg;
output [14:0] frame_address, out_address, in_address;
output [35:0] frame_data_in, frame_data_out;
output frame_we;
*/

//WIRES
wire frame_we;                      //Write enable for frame buffer
                                    //BRAM
wire [14:0] frame_address;          //Address for BRAM
wire [35:0] frame_data_out;         //Data out for BRAM. Note 36 pixels
                                    //even though only 32 are used.
                                    //This is to make adding a ZBT
                                    //easier

wire [14:0] in_address;             //Address for writing to buffer
wire [14:0] out_address;            //Address for reading from buffer
wire [14:0] current_address;        //Where current outreg data comes
                                    //from
wire [35:0] frame_data_in;          //Data from artist module minor
wire busy;                          //Busy signal - other modules pause
wire address_invalid;               //Goes high if the input coords are
                                    //outside the visible area
                                    //This is needed as memory usage
                                    //assumes addresses in 1024x768
                                    //region
wire [2:0] pixel;                   //Pixel output to XVGA


//REGISTERS

reg [14:0] reset_count;             //Resetting the screen takes
                                    //several clock cycles. Reset_count
                                    //keeps track of how far along it
                                    //is.

reg [31:0] outreg;                  //Buffer for next 32 pixel values

reg resetting=0;                    //Reset takes more than one clock
                                    //cycle, so use resetting     to
                                    //keep track of
                                    //whether a reset is taking place


//Frame Buffer BRAM (temp until ZBT arrives)
frame_buffer frame_buffer(
      frame_address,
      clock_65mhz,
      frame_data_in,
      frame_data_out,
      frame_we);
```

```
//SYNCHRONOUS LOGIC
always @ (posedge clock_65mhz)
begin

//RESET CODE
if (reset|clear_screen) begin              //See clear screen note above
      resetting <= 1;                      //Enter resetting mode
      reset_count <= 0;                    //Start counting how
                                           //many cycles have been
                                           //resetting for
      end
else begin
      if (resetting) begin
            reset_count <= reset_count + 1; //Work way through BRAM
            if (reset_count > 24574) resetting <= 0;  //Finished reset
      end

      //If are in the read portion the cycle put the memory output into
      //the 32 bit buffer register.
      else  if ((hcount%32) == 31) outreg <= frame_data_out[31:0];
      end
end

//COMBINATIONAL OUTPUTS
assign frame_address = busy ? out_address : in_address;
                                    //Busy muxes between the reading
                                    //and writing address

assign frame_we = (~busy & ~address_invalid);
                                    //Do not write when busy or when
                                    //input coords are outside the
                                    //valid screen area

assign pvsync = vsync;              //No need to change sync/blanking
assign phsync = hsync;
assign pblank = blank;

//If resetting put 0s into memory, otherwise use data input
assign frame_data_in =        (reset | resetting) ? 0
                              : slice_pixels;

//If resetting incremement address each clock cycle, otherwise write
//address = f(slice_y and slice_x)
assign in_address =           (reset | resetting) ? reset_count   :
                              {slice_y[9:0], slice_x[9:5]};

//Current address deals with reading from buffer
assign current_address =      {vcount[9:0], hcount[9:5]};

// If vcount/hcount is now in the region off the bottom of the screen,
// prepare to draw first element when they come round
assign out_address =          (current_address > 24574) ?   0

//If hcount is in the blanking region off the right
//of the screen, prepare to draw the first element of the next row
                              :(hcount[10:5] > 30) ?
                              {(vcount[9:0] + 1), 5'b00000}
```

```
//If neither of these conditions is true look at the next element along
                              :(current_address + 1);

//Address invalid signal stops invalid co-ordinates from contaminating
//the memory. Modules also use slice_y = 768 when they are 'resting'
//and do not want to write to the memory
assign address_invalid =            (reset|clear_screen|resetting) ? 0
                                    : ((slice_x > 1023)
                                          |(slice_y > 767));

//Busy signal tells higher modules that inputs will be ignored as the
//frame buffer reads out the next 32 bits
assign busy =                 (reset|clear_screen|resetting) ? 0 :
                              ((hcount%32) > 28);

assign pixel =                (reset|clear_screen|resetting) ? 0 :
                              ~(outreg[(31-hcount%32)]* 7);

endmodule
```

Artist Module Minor

```
module artist_module_minor (  artist_start, artist_done_p5,
                              start_row, n_pitch, n_octave, n_sprite,
                              n_sharp, n_dot,
                              clef,
                              slice_y_p5, slice_pixels_p5, busy,

                              /*
                              DEBUG I/O
                              sprite_address, sprite_data,
                              //slice_pitch_pixel, sprite_y_offset,
                              sprite_count,
                              initial_sprite_address,
                              current_sprite_address,
                              */
                              minor_state,

                              clock_65mhz, reset);

//ARTIST MAJOR FSM I/O
input artist_start;           //Start drawing signal
input [9:0] start_row;        //y-coord start of slice
input [2:0] n_octave;         //octave of slice
input [3:0] n_pitch;          //pitch of sice
input [5:0] n_sprite;         //sprite number to be used
input n_sharp;                //1 if note is a sharp
input n_dot;                  //1 if note is dotted

output artist_done_p5;        //Finished, new slice please!

//USER I/O
input [1:0] clef;             //0 bass
                              //1 alto
                              //2 treble
```

```
                              //3 tenor

//FRAME BUFFER I/O
input busy;                         //Tells artist module to pause
output[9:0] slice_y_p5;             //y-coord of row being outputted
output [31:0] slice_pixels_p5;      //pixel values for row

//GLOBAL I/O
input clock_65mhz, reset;

//DEBUG I/O
/*
output [12:0] sprite_address;
output [31:0] sprite_data;
//output [6:0] sprite_y_offset;
//output [8:0] slice_pitch_pixel;
output [7:0] sprite_count;
output [12:0] initial_sprite_address;
output [12:0] current_sprite_address;
*/
output [2:0] minor_state;

//PIPELINE REGISTERS
//Pipeline paths are shown in labbook
//_p means “pipelined” number refers to which layer of regs

//Piepline stage 5
reg [9:0] slice_y_p5;
reg [31:0] slice_pixels_p5;
reg [12:0] sprite_address_p5;
reg busy_p5;
reg [7:0] sprite_count_p5;
reg [31:0] sprite_data_p5;
reg [8:0] slice_pitch_pixel_p5;
reg state_done_p5;
reg artist_done_p5;
reg [5:0] n_sprite_p5;
reg artist_start_p5;

//Pipeline stage 3 (see lab book)
reg [12:0] initial_sprite_address_p3;
reg [12:0] current_sprite_address_p3;
reg [5:0] n_sprite_p3;
reg busy_p3;
reg [7:0] sprite_count_p3;
reg [31:0] sprite_data_p3;
reg [9:0] slice_y_p3;
reg [9:0] start_row_p3;
reg [8:0] slice_pitch_pixel_p3;
reg artist_start_p3;
reg n_sharp_p3;
reg n_dot_p3;

// WIRES
wire [31:0] sprite_data;      //Data output from sprite ROM
wire [12:0] sprite_address;   //Address for sprite memory
wire [12:0] current_sprite_address;
```

```
wire [8:0] slice_pitch_pixel;
wire artist_done;
wire state_done;
wire [9:0] slice_y;
wire [31:0] slice_pixels;
wire [4:0] slice_bottom;

//SPRITE MEMORY
// USING SPRITE_MEMORY.COE)
sprite_memory sprite_memory (
                              sprite_address_p5,
                              clock_65mhz,
                              sprite_data);

//PARAMETERISE STATES
//N.B ONLY TWO USED!
parameter minor_wait = 0;
parameter minor_sprite = 1;
parameter minor_slice = 2;
parameter minor_bar = 3;
parameter minor_blank = 4;

//REGS FOR MINOR FSM
reg [2:0] minor_state = 0;
reg [7:0] sprite_count = 0;
reg [12:0] initial_sprite_address;
reg [31:0] sharp_pixels;
reg [31:0] dot_pixels;
reg [31:0] sprite_pixels;


//PARAMETERISE CLEFS (might want to change order later)
parameter bass = 0;
parameter alto = 1;
parameter treble =2;
parameter tenor = 3;

//Slice bottom is what the 7*octave + n_pitch value is for the bottom
//of the slice.
assign slice_bottom =         (clef == bass)    ?     12
                              :(clef == alto)   ?     18
                              :(clef == treble) ?     24
                              : 16; //Last one is tenor

//SYNCHRONOUS LOGIC
always @ (posedge clock_65mhz)
begin
      //Reset Code
      if (reset) begin
            minor_state <= 0;
            sprite_count <= 0;
            end

      // Minor FSM state transition diagram implementation
      else begin
            case (minor_state)
```

```
            minor_wait:         if (artist_start_p5) minor_state
                                <=minor_sprite;
                                else minor_state <= minor_wait;
            default:            minor_state <= (state_done_p5)      ?
                                (minor_wait)       :     (minor_state);
                                //All other states return to wait once
                                //artist done is high

            endcase

      //Sequencer for Sprite Drawing Module
      if (minor_state == minor_wait) sprite_count <= 0;
      if (minor_state == minor_sprite) begin
            if (~busy_p5)
                  sprite_count <= sprite_count_p5 + 1;
            if (sprite_count_p5 == 2) initial_sprite_address <=
                  n_sprite_p5*80 + sprite_data_p5[7:0] -
                  slice_pitch_pixel_p5;
      end

      //Pipeline stage 3 (see lab book)
      //This mess just transfers signals up the pipeline chain
      {initial_sprite_address_p3, current_sprite_address_p3,
n_sprite_p3, busy_p3, sprite_count_p3, sprite_data_p3, slice_y_p3,
start_row_p3, slice_pitch_pixel_p3, artist_start_p3, n_sharp_p3,
n_dot_p3}
      <= {initial_sprite_address, current_sprite_address, n_sprite,
busy, sprite_count, sprite_data, slice_y, start_row, slice_pitch_pixel,
artist_start, n_sharp, n_dot};

      //Pipeline stage 5 (see lab book)
      {slice_y_p5, slice_pixels_p5, sprite_address_p5, busy_p5,
sprite_count_p5, sprite_data_p5, slice_pitch_pixel_p5, state_done_p5,
artist_done_p5, n_sprite_p5, artist_start_p5}
      <= {slice_y_p3, slice_pixels, sprite_address, busy,
sprite_count_p3, sprite_data_p3, slice_pitch_pixel_p3, state_done,
artist_done, n_sprite_p3, artist_start_p3};

      end
end

//COMBINATIONAL OUTPUTS
//Slice pitch pixel is the pixel row that the sprite should be centered
//on 140 is the number of pixels high a slice is, slice_bottom is a
//function of clef
assign slice_pitch_pixel
            = 140 - (n_octave*35 + 5*n_pitch - 5*slice_bottom);
//Orginally above line was like this but had higher combinational
//delay!
//140 - (n_octave_p1*7 + n_pitch_p1 - slice_bottom_p1) * 5

assign current_sprite_address
            = (initial_sprite_address + sprite_count - 3);

assign sprite_address
            = (sprite_count_p3 <  2) ? (n_sprite_p3 * 80) :
            (sprite_count_p3 <  4) ? (initial_sprite_address_p3) :
```

```
            (busy_p3)               ? (current_sprite_address_p3 - 1):
            (current_sprite_address_p3);

//Purpose of '768' is to tell frame buffer that output is now invalid
//so don't alter the memory
assign slice_y
      = ((minor_state == 1)&(sprite_count > 3)&(~busy))
      ? start_row + sprite_count - 4 : 768;


//Combinational logic to support sharps, dots
always @ ( current_sprite_address_p3, n_sprite_p3, sprite_data_p3,
sprite_count_p3,slice_y_p3, start_row_p3, slice_pitch_pixel_p3,
n_sharp)
begin
      //Sharp pixels. Puts a picture of a sharp sign onto sharp_pixels
      case ((slice_y_p3 - start_row_p3) - (slice_pitch_pixel_p3 + 5))
      1:    sharp_pixels =    32'b00000000000000000000010000001000;
      2:    sharp_pixels =    32'b00000000000000000000010000001000;
      3:    sharp_pixels =    32'b00000000000000000011111111111111;
      4:    sharp_pixels =    32'b00000000000000000000010000001000;
      5:    sharp_pixels =    32'b00000000000000000000010000001000;
      6:    sharp_pixels =    32'b00000000000000000000010000001000;
      7:    sharp_pixels =    32'b00000000000000000011111111111111;
      8:    sharp_pixels =    32'b00000000000000000000010000001000;
      9:    sharp_pixels =    32'b00000000000000000000010000001000;
      default:          sharp_pixels =
      32'b00000000000000000000000000000000;
      endcase

      //Puts a dot onto dot_pixels
      case ((slice_y_p3 - start_row_p3) - (slice_pitch_pixel_p3 + 2))
      1:    dot_pixels = 32'b00000000000000000000000000111000;
      2:    dot_pixels = 32'b00000000000000000000000001111100;
      3:    dot_pixels = 32'b00000000000000000000000000111000;
      default:    dot_pixels = 32'b00000000000000000000000000000000;
      endcase

//Stave lines
if    (((sprite_count_p3 - 5) == 60)||
       ((sprite_count_p3 - 5) == 70)||
       ((sprite_count_p3 - 5) == 80)||
       ((sprite_count_p3 - 5) == 90)||
       ((sprite_count_p3 - 5) == 100)) sprite_pixels = 32'hffffffff;

//If sprite address is in a region where it is looking at the right
sprite it uses the sprite data, otherwise blanks
else  sprite_pixels
      =     (((current_sprite_address_p3 - 1)> (n_sprite_p3 *80))
            &&((current_sprite_address_p3 ) < (n_sprite_p3*80 + 80))) ?
            sprite_data_p3 :
            0;


end

assign slice_pixels
```

```
            =     (n_sharp_p3 && n_dot_p3) ?
            (sharp_pixels | sprite_pixels | dot_pixels)
            :(n_sharp_p3) ? (sharp_pixels | sprite_pixels)
            :(n_dot_p3) ? (dot_pixels | sprite_pixels)
            :sprite_pixels;


assign state_done = (sprite_count_p3 == 142) ? 1 : 0;
assign artist_done = (sprite_count_p3 == 143) ? 1 : 0;

endmodule
```

Artist Module Major

```
//Artist Module Major FSM
//Takes an input of three slices in parallel from graphics controller
//and passes them in series to the minor FSM. Minor and major FSM
//coordinate with artist_start and artist_done. The major FSM
//also computes values for the postioning of each slice on the screen
//(it only receives that a particular slice represents the nth note).
//This requires use of a divider. The divider module takes roughly
//12 clock cycles to compute valid values of alice_x and start_row.

module artist_module_major(   n0_octave, n0_pitch, n0_sprite,
                              n0_countslice, n0_sharp, n0_dot,
                              n1_octave, n1_pitch, n1_sprite,
                              n1_countslice, n1_sharp, n1_dot,
                              n2_octave, n2_pitch, n2_sprite,
                              n2_countslice, n2_sharp, n2_dot,
                              graphics_ready,
                              display_ready,
                              slice_x_p5, start_row_p0,
                              n_pitch_p0, n_octave_p0, n_sprite_p0,
                              n_sharp_p0, n_dot_p0,
                              artist_start_p0, artist_done,

                              /*
                              //Debug
                              //remd1, remd2, dividend, divisor1,
                              divisor2, quot1, quot2,
                              start_divide,
                              //n_countslice, old_n_countslice,
                              //divide_ready1, divide_ready2,
                              divide_ready_count,
                              */
                              major_state,
                              major_toggle,


                              clock_65mhz, reset);

//GRAPHICS CONTROLLER I/O
// n0-> n2 are the next three slices to be drawn
input [2:0] n0_octave, n1_octave, n2_octave;          //octave number
input [3:0] n0_pitch, n1_pitch, n2_pitch;             //natural pitch
                                                      //(does not
```

```
                                                            //include sharps
                                                            //or flats)
input [5:0] n0_sprite, n1_sprite, n2_sprite;                //sprite number
input n0_sharp, n1_sharp, n2_sharp;                         //1 if sharp
input n0_dot, n1_dot, n2_dot;                               //1 if dotted
input [11:0] n0_countslice, n1_countslice, n2_countslice;
                                            //note number of new slices

input graphics_ready;                       //Graphics controller's
                                            //command to start drawing
                                            //(A single clock cycle
                                            //pulse)

output display_ready;                       //Lets graphics controller
                                            //know when it can change n0-
                                            //n2 and send a new
                                            //graphics_ready
                                            //High on ready
//Global IO
input clock_65mhz, reset;

//Artist_module_major to frame_buffer I/O
output [10:0] slice_x_p5;             //Slice starting x coordinate (NB
                                      //always divisible by 32)

//Artist_module_major - > Artist_module_minor I/O
input artist_done;                    //Pulses high when minor FSM is
                                      //done drawing

output artist_start_p0;               //Pulses high for minor FSM to
                                      //start drawing
output [9:0] start_row_p0;            //Starting y coordinate for slice
output [2:0] n_octave_p0;             //n_ prefix indicates slice being
                                      //drawn right now
output [3:0] n_pitch_p0;
output [5:0] n_sprite_p0;
output n_sharp_p0;
output n_dot_p0;


//DEBUG IO (Left in so that it can be easily unremmed to show operation
of module)
//output [11:0] remd1, remd2, dividend, divisor1, divisor2, quot1,
quot2;
//output start_divide;
//output [11:0] n_countslice, old_n_countslice;
//input start_divide;
//output divide_ready1, divide_ready2;
//output [1:0] divide_ready_count;
output [1:0] major_state;
output major_toggle;

//PIPELINE REGISTERS
reg artist_start_p0;
reg [9:0] start_row_p0;
reg [2:0] n_octave_p0;
reg [3:0] n_pitch_p0;
```

```
reg [5:0] n_sprite_p0;
reg n_sharp_p0;
reg n_dot_p0;
reg [10:0] slice_x_p3;
reg [10:0] slice_x_p5;


// DECLARE REGISTERS FOR MAJOR FSM
reg [1:0] major_state = 0;          //State variable for major FSM
reg [1:0] old_major_state=3;        //Needed for outputs on state
                                    //transitions (start_divide)
//reg [1:0] divide_ready_count = 0; //Needed because module is waiting
//on two dividers to give valid outputs
//therefore need to count ready pulses

reg major_toggle = 0;
reg [9:0] cludge_count = 0;         //Knocks FSM out of stuck states
reg old_dividers_ready = 1;


// DECLARE OUTPUTS FOR MAJOR FSM
// (Assigned combinationally)
wire [9:0] start_row;               //Starting y coordinate for slice n
wire [10:0] slice_x;                //Starting x coordinate for slice n
wire artist_start;                  //Signal to minor FSM that all
                                    //inputs are valid, begin drawing
wire display_ready;                 //Signal to graphics controller
                                    //that major FSM can cope with a
                                    //new set of ns
wire divide_ready1, divide_ready2;  //Ready signals from dividers for
                                    //slice_x, start_row
wire dividers_ready;
assign dividers_ready = ((divide_ready1) && (divide_ready2));

// Declare outputs for major FSM -> divider inputs
wire [11:0] dividend;               //Dividend is the same for both
(n_countslice)
wire [11:0] divisor1, divisor2;     //Divisors for d1 and d2
wire [11:0] remd1, remd2;           //The remainder outputs. The
                                    //dividers compute the quotients as
                                    //wellbut only the remainders are
                                    //needed

//n's parameters need to be regs because they are assigned within an
always block, but do not end up latched
reg [2:0] n_octave;
reg [3:0] n_pitch;
reg [5:0] n_sprite;
reg [11:0] n_countslice=0;
reg n_sharp;
reg n_dot;

//Display parameters. Parameterised to make changes easier and code
//more readable.
//These parameters are needed for the computation of slice_x and
//slice_y
parameter slices_per_row = 30;
```

```
parameter pixels_per_slice = 32;
parameter slices_per_screen = 120;
parameter pixels_per_row = 160;
parameter y_margin = 64;
parameter x_margin = 32;

//Parameterise states
//States for major FSM
parameter major_wait = 0;
parameter major_draw_n2 = 1;
parameter major_draw_n1 = 2;
parameter major_draw_n0 = 3;

//Originally had start_divide triggered off changes in n_countslice but
//that gave problems when the state changed
//but n_countslice didn't (although that should never happen...) - need
//the artist_start signal which needs the
//start_divide signal

assign start_divide = ((old_major_state != major_state)&(major_state !=
major_wait));

//SYNCHRONOUS LOGIC
always @ (posedge clock_65mhz)
begin
      //Reset code
      if (reset) begin
            major_state <= 0;
            old_major_state <=0;
            //divide_ready_count <=0;
            old_dividers_ready <= 1;
      end

      //Major FSM state transistion diagram

      else begin
            //Control loops around states- see lab book page 37
            case (major_state)
            major_wait:       major_state <=    (graphics_ready)  ?
                              major_draw_n2     :     major_wait;
            major_draw_n2:    major_state <=
                              (artist_done|(cludge_count == 500))?
                              major_draw_n1     :     major_draw_n2;
            major_draw_n1:    major_state <=
                              (artist_done|(cludge_count == 500))?
                              major_draw_n0     :     major_draw_n1;
            major_draw_n0:    major_state <=
                              (artist_done|(cludge_count == 500))?
                              major_wait  :     major_draw_n0;
            endcase

      // Other FSM regs
      // Start_divide triggers on state transitions, so need //level to
      pulse on state changes
      old_major_state <= major_state;
      old_dividers_ready <= dividers_ready;
```

```
        //Cludge count is a monstrosity created to overcome an apparently
        //random bug that soemtimes traps
        //major FSM in a drawing state. If no artist_done cycle has been
        //received after 800 clock cycles
        //(more than enough time for minor to do its stuff) it skips to
        //the next state regardless


        if (start_divide) cludge_count <= 1;
        else cludge_count <=    (cludge_count == 0)           ?     0
                                :(cludge_count == 500)  ?     0     :
                                (cludge_count + 1);


        //Major toggle is a useful debug output that toggles an led
        //whenever major FSM cycles
        major_toggle <= (artist_done & (major_state == major_draw_n0)) ?
        ~major_toggle : major_toggle;
        // Divide_ready_count keeps track of how many dividers have
        //finished computing
        /*
        // Artist_start triggers when divide_ready_count == 2
        divide_ready_count <=   (divide_ready_count == 2)           ?
        0
                                      :(divide_ready1 && divide_ready2)
        ?     2
                                      :(divide_ready1 | divide_ready2)
        ?     divide_ready_count +1
                                      :divide_ready_count;
        */

        //PIPELINING
        artist_start_p0 <= artist_start;
        start_row_p0 <= start_row;
        n_octave_p0 <= n_octave;
        n_pitch_p0 <= n_pitch;
        n_sprite_p0 <= n_sprite;
        n_sharp_p0 <= n_sharp;
        n_dot_p0 <= n_dot;
        slice_x_p3 <= slice_x;
        slice_x_p5 <= slice_x_p3;

        end

end

// MAJOR FSM OUTPUTS
// Display ready is high when major FSM is in the wait state
assign display_ready =
            ((major_state == major_wait)&(old_major_state
            != major_state)); //Only accept inputs in wait state

//Always block makes assigning multiple n variables easier.
always @ (  major_state, n1_pitch, n1_octave, n1_sprite,
            n1_countslice,n2_pitch, n2_octave, n2_sprite,
            n2_countslice, 0_pitch, n0_octave, n0_sprite,
            n0_countslice, n2_sharp, n2_dot, n1_sharp,
```

```
            n1_dot, n0_sharp, n0_dot)
begin
      case (major_state)
      //This case statement muxes n0 -> n2 into n
      major_wait: begin
      //Originally all values here were x, but this caused problems
      //with simulation
                  n_pitch = 4'b0;
                  n_octave = 3'b0;
                  n_sprite = 6'b0;
                  n_countslice = 12'b0;
                  n_sharp = 0;
                  n_dot = 0;
                  end

      major_draw_n2:    begin
                  n_pitch = n2_pitch;
                  n_octave = n2_octave;
                  n_sprite = n2_sprite;
                  n_countslice = n2_countslice;
                  n_sharp = n2_sharp;
                  n_dot = n2_dot;
                  end

      major_draw_n1:    begin
                  n_pitch = n1_pitch;
                  n_octave = n1_octave;
                  n_sprite = n1_sprite;
                  n_countslice = n1_countslice;
                  n_sharp = n1_sharp;
                  n_dot = n1_dot;
                  end

      major_draw_n0:    begin
                  n_pitch = n0_pitch;
                  n_octave = n0_octave;
                  n_sprite = n0_sprite;
                  n_countslice = n0_countslice;
                  n_sharp = n0_sharp;
                  n_dot = n0_dot;
                  end
                  //No need for default - all possible states are
                  //used
      endcase
end

//Dividers
//Set up divisors and dividend
assign dividend = n_countslice;
assign divisor1 = slices_per_row;
assign divisor2 = slices_per_screen;

//Artist_start waits for both dividers to finish (i.e. start_row and
//x_slice valid)
//assign artist_start = (divide_ready_count == 2);
assign artist_start = ((dividers_ready)&&(!old_dividers_ready));
```

```
//Computation of starting x coordinate for slice
assign slice_x = remd1 * pixels_per_slice + x_margin;

//Computation of starting y coordinate for slice. Orginally this would
//have required another modulo division
//but found a way around it using muxes.
assign start_row =      (remd2 < slices_per_row)
            ?     y_margin :
            (remd2 > (slices_per_screen - slices_per_row -1))     ?
            pixels_per_row * 3 + y_margin:
            (remd2 > (slices_per_screen - 2*slices_per_row -1))   ?
            pixels_per_row * 2 + y_margin:
            (pixels_per_row + y_margin);


//SUB MODULES -> DIVIDERS

easy_divider d1(  //quot1,
                        remd1,
                        divide_ready1,
                        dividend,
                        divisor1,
                        start_divide,
                        reset,
                        clock_65mhz);

easy_divider d2(  //quot2,
                        remd2,
                        divide_ready2,
                        dividend,
                        divisor2,
                        start_divide,
                        reset,
                        clock_65mhz);

endmodule
```

Graphics Controller

```
//Use postscript _p for pipelined outputs

module graphics_controller_simple ( p_octave, p_pitch, p_startbeat,
                                    p_duration, prov_ready,
                                    n0_octave_p, n0_pitch_p,
                                    n0_sprite_p, n0_countslice_p,
                                    n0_sharp_p, n0_dot_p,
                                    n1_octave_p, n1_pitch_p,
                                    n1_sprite_p, n1_countslice_p,
                                    n1_sharp_p, n1_dot_p,
                                    n2_octave_p, n2_pitch_p,
                                    n2_sprite_p, n2_countslice_p,
                                    n2_sharp_p, n2_dot_p,
                                    clef, timesig,

                                    graphics_ready_p, display_ready,
                                    clear_screen,
```

```
                                    //playback,
                                    //Debug I/O begins
                                    /*
                                    controller_state,
                                    n2_sprite_old, n1_sprite_old,
                                    */
                                    go_bar, bar2, bar1,

                                    //Debug I/O ends
                                    reset, clock_65mhz);

// Adam's Side (audio processing) I/O
input [2:0] p_octave;         // Octave of new note (0-6 inlcusive)
input [3:0] p_pitch;          // Pitch of new note (0-11 inclusive)

/*
      Table of score_pitch values
      ________________________________________________
      |3:   C                   | 10:      G                  |
      |4:   Csharp == Dflat     | 11:      Gsharp == Aflat   |
      |5:   D                   | 0: A                     |
      |6:   Dsharp == Eflat     | 1: Asharp == Bflat   |
      |7:   E                   | 2: B                     |
      |8:   F                   |                              |
      |9:   Fsharp == Gflat     |                              |
      |______________________   |__________________________|
*/

input [11:0] p_startbeat;
                                    // Countslice  0 -> 4000 (overkill,
                                    //allows a 20minute 200bpm piece
                                    // to be recorded (N.B most House
                                    //music < 140 BPM so this is safe
                                    //for instrumental!)
input [3:0] p_duration;             // 0-11 inclusive
/*
      Table of score_duration values
      ________________________________________________
      |0:   1/32                  |6: 1/32 dotted   |
      |1:   1/16            |7: 1/16 dotted   |
      |2:   1/8                   |8: 1/8 dotted    |
      |3:   1/4                   |9: 1/4 dotted    |
      |4:   1/2                   |10:      1/2 dotted    |
      |5:   1                     |11:      1 dotted            |
      |_____________________|_________________________|
*/

input prov_ready;                   //Pulses high when new score e
                                    //element is available

//Major artist FSM I/O
output [2:0] n0_octave_p, n1_octave_p, n2_octave_p;
                  //octave number
output [3:0] n0_pitch_p, n1_pitch_p, n2_pitch_p;
                  //natural pitch (does not include sharps or flats)
output [5:0] n0_sprite_p, n1_sprite_p, n2_sprite_p;
                  //sprite number
```

```
output n0_dot_p, n1_dot_p, n2_dot_p;
                  //0 for not dotted, 1 for dotted
output n0_sharp_p, n1_sharp_p, n2_sharp_p;
                        //0 = natural
                        //1 = sharp
output [11:0] n0_countslice_p, n1_countslice_p, n2_countslice_p;
                        //note number of new slices
output graphics_ready_p;
            //Graphics controller's command to start drawing
            //(A single clock cycle pulse)
input display_ready;

//User inputs
input [1:0] clef;       //See memory offsets below for meaning of clef,
                        //timesig
input [2:0] timesig;

//Frame buffer I/O
output clear_screen;
wire clear_screen;

// Global I/O
input reset;
input clock_65mhz;
/*
// Debug I/O
output [2:0] controller_state;
output [5:0] n1_sprite_old, n2_sprite_old;
*/
output go_bar, bar2, bar1;

//FSM Output Declaration
reg [2:0] n0_octave=0, n1_octave=0, n2_octave=0;
      //octave number
reg [3:0] n0_pitch=0, n1_pitch=0, n2_pitch=0;
      //natural pitch (does not include sharps or flats)
reg [5:0] n0_sprite=0, n1_sprite=0, n2_sprite=0;
      //sprite number
reg n0_dot=0, n1_dot=0, n2_dot=0;
      //0 for not dotted, 1 for dotted
reg n0_sharp = 0, n1_sharp=0, n2_sharp=0;
      //0 = natural
      //1 = sharp

wire [11:0] n1_countslice, n2_countslice; //note number of new slices
wire [11:0] n0_countslice;
wire graphics_ready;

reg p_ready;


//FSM Pipelined Output Declaration
reg [2:0] n0_octave_p=0, n1_octave_p=0, n2_octave_p=0;
      //octave number
reg [3:0] n0_pitch_p=0, n1_pitch_p=0, n2_pitch_p=0;
      //natural pitch (does not include sharps or flats)
```

```
reg [5:0] n0_sprite_p=0, n1_sprite_p=0, n2_sprite_p=0;
      //sprite number
reg n0_dot_p=0, n1_dot_p=0, n2_dot_p=0;
      //0 for not dotted, 1 for dotted
reg n0_sharp_p = 0, n1_sharp_p=0, n2_sharp_p=0;                   //0 =
natural

      //1 = sharp
reg [11:0] n1_countslice_p, n2_countslice_p;    //note number of new
slices
reg [11:0] n0_countslice_p;
reg graphics_ready_p;

reg [2:0] n1_octave_old = 0, n2_octave_old = 0, n0_octave_old=0;
reg [3:0] n1_pitch_old = 0, n2_pitch_old = 0, n0_pitch_old=0;
reg [5:0] n1_sprite_old =0 , n2_sprite_old = 0, n0_sprite_old =0;
reg n1_dot_old = 0, n2_dot_old = 0, n0_dot_old = 0;

reg n1_sharp_old = 0, n2_sharp_old = 0, n0_sharp_old;

//Duration registers are for working out what goes barred. Set them to
//7 at first (impossible value)
//so that don't get bars appearing randomly on a reset
reg [2:0] n2_nduration_old=7, n1_nduration_old=6, n0_nduration_old=5;
reg [2:0] n2_nduration=7, n1_nduration=6, n0_nduration=5;

reg [11:0] n0_countslice_old = 0, n1_countslice_old = 0,
n2_countslice_old = 0;  //note number of new slices

//Parameterise states
parameter controller_wait = 0;
parameter controller_setup = 1;
parameter controller_dsprite = 2;
parameter controller_dbarred = 3;
parameter controller_drest = 4;

//Parameterise Sprite Memory Locations
//See also spreadsheet spritememoryspread.xls

parameter memory_clefs = 0;
/* CLEFS OFFSETS
Offset      Type
0           Bass
1           Alto
2           Treble
3           Tenor
*/
parameter memory_timesig = 4;
/* TIME SIGNATURE OFFSETS
Offset      Type
0           4 4
1           2 2
2           2 4
3           3 4
4           6 8
*/
parameter   memory_rests = 8;
```

```
/* RESTS OFFSET
Offset      Type
0           1/32
1           1/16
2           1/8
3           1/4
4           1/2
5           1
*/
parameter memory_notes_p = 15;
/* NOTES (p) OFFSET
As for rests
*/
parameter memory_notes_d = 21;
/* NOTES (d) OFFSET
As for rests
*/
parameter memory_bar_p = 27;
/* BARRED NOTES (p) OFFSET
Offset      Type              Offset      Type
0           1/32 left         6           1/8 left
1           1/32 middle 7           1/8 middle
2           1/32 right  8           1/8 right
3           1/16 left
4           1/16 middle
5           1/16 right
*/
parameter memory_bar_d = 36;
/* BARRED NOTES (d) OFFSET
Offset      Type              Offset      Type
0           1/32 left         6           1/8 left
1           1/32 middle 7           1/8 middle
2           1/32 right  8           1/8 right
3           1/16 left
4           1/16 middle
5           1/16 right
*/
parameter memory_blank = 45;
//Contains nothing
parameter memory_special = 46;
//Things that might be useful for debugging
//0 something to show that a barred note should have been drawn here
//1 sprite assigned in non-drawing states. Hopefully will never see
this!
//2 sprite assigned if rest is longer than a whole note
//3 perfect pitch logo!

//Parameterise clefs (might want to change order later)
parameter bass = 0;
parameter alto = 1;
parameter treble = 2;
parameter tenor = 3;

//Parameterise the octave and pitch for "static" objects (rests and
clefs)
wire [2:0] static_octave;
wire [2:0] static_pitch;
```

```
assign static_octave =   (clef == tenor) ?      3
                              :(clef == alto)  ?      4
                              :(clef == bass)  ?      3
                              :(clef == treble)?  4
                              : 4;

assign static_pitch =   (clef == tenor) ? 2
                              :(clef == alto) ? 0
                              :(clef == bass) ? 1
                              :(clef == treble)?  6
                              :6;

//Registers for graphics controller FSM
reg [2:0] controller_state = 0;
reg [2:0] old_controller_state= 0;
reg [6:0] slice_count=2;                  //Slice count keeps track of
how many slices have been drawn to the screen
                                          //this page. It is needed
because the number notes does not equal the
                                          //number of slices drawn
(rests, clefs)

//CONTROL SIGNALS

//Work out p_npitch and p_nduration
reg [2:0] p_npitch;
//p_npitch takes the sharp information out of p_pitch and assigns it to
a different variable (p_sharp)
//This is useful as it allows p_npitch to be used as an offset for
calculating note positions on the stave
//(f# should be on the same line as f)

always @ (p_pitch)
case  (p_pitch)
0:    p_npitch = 0;
1:    p_npitch = 0;
2:    p_npitch = 1;
3:    p_npitch = 2;
4:    p_npitch = 2;
5:    p_npitch = 3;
6:    p_npitch = 3;
7:    p_npitch = 4;
8:    p_npitch = 5;
9:    p_npitch = 5;
10:   p_npitch = 6;
11:   p_npitch = 6;
default: p_npitch = 0;
endcase

assign p_sharp =  ((p_pitch == 1)|(p_pitch == 4)|(p_pitch ==
6)|(p_pitch == 9)|(p_pitch == 11)) ?
                        1 : 0;

//Do a similar thing to duration: dot does not determine type of
sprite.
wire  [2:0] p_nduration;
```

```
assign p_nduration =  (p_duration > 5) ? (p_duration - 6) : p_duration;
//Deal with dots from the graphics side by whacking in a dot in a
similar way to the lines (except will
//be based on y_offset.) From the controller point of view does not
alter sprite selected but will send
//a one bit signal saying "draw a dot"
wire p_dot;
assign p_dot        =  (p_duration >5);

wire rest=0;      //Temporary
wire go_bar, bar2, bar1;

assign go_bar =         ((p_octave == n2_octave_old)&&(p_npitch ==
n2_pitch_old)&&(p_nduration == n2_nduration_old)&&(p_nduration < 3));
assign bar2 =           ((n1_octave_old ==
n2_octave_old)&&(n1_pitch_old == n2_pitch_old) && (n1_nduration_old ==
n2_nduration_old)&&(n2_nduration < 3));
assign bar1 =           ((n1_octave_old ==
n0_octave_old)&&(n1_pitch_old == n0_pitch_old) && (n1_nduration_old ==
n0_nduration_old)&&(n1_nduration <3));

//Code above makes separate beams for 1/32 and 1/16 for example. Code
below puts them in the same beam with different flags
/*
assign go_bar =   ((p_octave == n2_octave_old)&&(p_npitch ==
n2_pitch_old)&&(p_nduration < 3));
assign bar2 =           ((n1_octave_old ==
n2_octave_old)&&(n1_pitch_old == n2_pitch_old) &&(n2_nduration < 3));
assign bar1 =           ((n1_octave_old ==
n0_octave_old)&&(n1_pitch_old == n0_pitch_old) &&(n1_nduration <3));
*/


wire new_state;
assign new_state = (controller_state != old_controller_state);


//STATE TRANSISTION/SEQUENCER
always @ (posedge clock_65mhz)
begin
//Only accept the ready signal if the note passed is within the
drawable range of the clef
//Do this for treble clef only initially, then expand)
p_ready <= (((p_octave < 8)&&(p_pitch<2))||((p_octave>2)&&(p_pitch>1)))
? prov_ready : 0;
//Reset code
if (reset) begin
      controller_state <= 0;
      old_controller_state <= 0;
      slice_count <= 2;
      end
else begin
      old_controller_state <= controller_state;

      //State Transistion Diagram
      case (controller_state)
      controller_wait:  begin
```

```
                              if (!p_ready) controller_state <=
controller_wait;
                                    else begin
                                          if (slice_count == 2)
controller_state <= controller_setup;
                                          else if (go_bar)
controller_state <= controller_dbarred;
                                                else controller_state
<= controller_dsprite;
                                          end
                              end

      controller_setup: controller_state <= (display_ready) ?
      controller_dsprite : controller_setup;
      controller_dsprite:     controller_state <=     (display_ready) ?
      controller_wait : controller_dsprite;
      controller_dbarred:     controller_state <= (display_ready) ?
      controller_wait : controller_dbarred;
      /*
      controller_drest: controller_state <= (display_ready) ?
      controller_wait: controller_drest;
      */
      default:                controller_state <= controller_wait;

      endcase

      //Sequencer (sort of)
      if (slice_count > 119) slice_count <= (slice_count - 118);
      else if (display_ready)
            case (controller_state)
            controller_setup:       slice_count <= 3;
            controller_dsprite:           slice_count <= slice_count +
1;
            controller_drest:       slice_count <= slice_count + 2;
            controller_dbarred:           slice_count <= slice_count +
1;
            default:                      slice_count <= slice_count;
            endcase

      //Backup n1, n2 once they have been reassigned (discard n0)
      if (((display_ready)&(controller_state == controller_dsprite))|
            ((display_ready)&(controller_state == controller_setup))|
            ((display_ready)&(controller_state == controller_dbarred)))
begin
      {n1_countslice_old, n1_octave_old, n1_pitch_old, n1_sprite_old,
n1_dot_old, n1_sharp_old, n1_nduration_old}
            <= {n1_countslice, n1_octave, n1_pitch, n1_sprite, n1_dot,
n1_sharp, n1_nduration};
      {n2_countslice_old, n2_octave_old, n2_pitch_old, n2_sprite_old,
n2_dot_old, n2_sharp_old, n2_nduration_old}
            <= {n2_countslice, n2_octave, n2_pitch, n2_sprite, n2_dot,
n2_sharp, n2_nduration};
      {n0_countslice_old, n0_octave_old, n0_pitch_old, n0_sprite_old,
n0_dot_old, n0_sharp_old, n0_nduration_old}
            <= {n0_countslice, n0_octave, n0_pitch, n0_sprite, n0_dot,
n0_sharp, n0_nduration};
      end
```

```
	//Pipeline
	{n0_octave_p, n0_pitch_p, n0_sprite_p, n0_countslice_p,
n0_sharp_p, n0_dot_p,
	n1_octave_p, n1_pitch_p, n1_sprite_p, n1_countslice_p,
n1_sharp_p, n1_dot_p,
	n2_octave_p, n2_pitch_p, n2_sprite_p, n2_countslice_p,
n2_sharp_p, n2_dot_p,
	graphics_ready_p} <=
	{n0_octave, n0_pitch, n0_sprite, n0_countslice, n0_sharp, n0_dot,
	n1_octave, n1_pitch, n1_sprite, n1_countslice, n1_sharp, n1_dot,
	n2_octave, n2_pitch, n2_sprite, n2_countslice, n2_sharp, n2_dot,
	graphics_ready};

	end
end



//FSM OUTPUTS


//n1 and n2 always follow on from n0
assign n2_countslice = slice_count;
assign n1_countslice = (n2_countslice < 1) ? 119 : n2_countslice - 1;
assign n0_countslice = (n2_countslice < 2) ? (119- n2_countslice) :
n2_countslice - 2;


//Graphics ready
assign graphics_ready =       ((new_state)&
                              ((controller_state == controller_setup)|
                               (controller_state ==
controller_dsprite)|
                               (controller_state ==
controller_dbarred)|
                               (controller_state ==
controller_drest)));

always @ (controller_state,
            timesig,
            clef, p_npitch, p_octave, p_nduration, n2_sprite_old,
n1_sprite_old, p_dot,
            n2_dot_old, n1_dot_old, p_sharp, n2_sharp_old,
n1_sharp_old, p_startbeat, n2_countslice,
            n2_pitch_old, n2_octave_old, n1_pitch_old, n1_octave_old,
            n2_nduration_old, n1_nduration_old, bar1, bar2
            ) begin

//n_sprite, n_dot, n_sharp signals
case (controller_state)
controller_setup:      begin
                              n2_sprite = memory_timesig + timesig;
                              n1_sprite = memory_clefs + clef;
                              n0_sprite = memory_special + 2;

                              //Setup sprites do not have "duration"
but set them all different (and with n2 having an
```

```
                              //"impossible" value so that don't get
go_bar signal initially.
                              n2_nduration = 7;
                              n1_nduration = 6;
                              n0_nduration = 5;

                              n2_sharp = 0;
                              n1_sharp = 0;
                              n0_sharp = 0;

                              n2_dot = 0;
                              n1_dot = 0;
                              n0_dot = 0;


                              end

controller_dsprite:           begin
                              //New sprite. First decide whether it is
a d or p note (tail pointing up or down)
                              //This depends on its position on the
stave, which depends on its pitch, octave
                              //and the clef. Then decide on which
sprite to use. This is a function of its duration
                              n2_sprite =
                                    (clef == tenor) ? //Tenor clef is
the irritating one,
                                                            //cannot be
assigned with multiplication)
                                     (((p_npitch + p_octave * 7)<28)
      ? (p_nduration + memory_notes_d)

            : (p_nduration + memory_notes_p))
                                    :((((p_npitch + p_octave *
7)<(24+clef*6))? (p_nduration + memory_notes_d)

            : (p_nduration + memory_notes_p));

                              //Shift previous sprites one space back
                              n1_sprite = n2_sprite_old;
                              n0_sprite = n1_sprite_old;

                              //Deal with dots and sharps
                              n2_dot = p_dot;
                              n1_dot = n2_dot_old;
                              n0_dot = n1_dot_old;

                              n2_sharp = p_sharp;
                              n1_sharp = n2_sharp_old;
                              n0_sharp = n1_sharp_old;

                              n2_nduration = p_nduration;
                              n1_nduration = n2_nduration_old;
                              n0_nduration = n1_nduration_old;
                              end

controller_drest:       begin
```

```
                              //If there is a rest procedure is similar
except are assigning two slices - the most recent
                              //contains the new note and the one
behind it contains the rest

                              //Note sprite assigned as before
                              n2_sprite =
                                    (clef == tenor) ? //Tenor clef is
the irritating one,
                                                            //cannot be
assigned with multiplication)
                                     (((p_npitch + p_octave * 7)<28)
      ? (p_nduration + memory_notes_d)

            : (p_nduration + memory_notes_p))
                                    :((((p_npitch + p_octave *
7)<(24+clef*6))? (p_nduration + memory_notes_d)

            : (p_nduration + memory_notes_p));

                              //Assigning the rest sprite is similar to
assigning a note sprite, the differences are
                              //that there is no d/p decision to make
and that the rest will always be assigned to
                              //the same position on the stave. When
more than 32 beats have passed then the rest cannot
                              //be represented with a single rest
sprite, so the device uses a special sprite
                              //(probably a break)
                              n1_sprite = ((p_startbeat -
n2_countslice) < 33) ?
                                                        (memory_rests +
(p_startbeat - n2_countslice))
                                                        :(memory_special
+ 2);
                              n0_sprite = n2_sprite_old;

                              //Deal with dots (no dotted/sharp rests!)
                              n2_dot = p_dot;
                              n1_dot = 0;
                              n0_dot = n2_dot_old;

                              n2_sharp = p_sharp;
                              n1_sharp = 0;
                              n0_sharp = n2_sharp_old;

                              //THIS IS PROBABLY WRONG - REVISE WHEN
YOU DO RESTS
                              n2_nduration = p_nduration;
                              n1_nduration = n2_nduration_old;
                              n0_nduration = n1_nduration_old;
                              end

controller_dbarred:           begin
                              //Deals with sprites for barred notes

                              //First off decide on p or d type
```

```
                              if    ((((clef==tenor)&(p_npitch +
p_octave * 7)<28))
                                    |((clef!=tenor)&((p_npitch +
p_octave * 7)<(24+clef*6))))
                                    begin //d type note
                                          if (bar1 & bar2) begin
//last three notes were all same duration etc.
                                          //n2 will be a "right" type
                                          n2_sprite = (3* p_nduration +
2 + memory_bar_d) ;
                                          //n1 will be a "middle" type
                                          n1_sprite = (3*
n2_nduration_old + 1 + memory_bar_d);
                                          //n0 is also a "middle"type
                                          n0_sprite = (3*
n1_nduration_old + 1 + memory_bar_d);
                                          end

                                          else if (!bar1 & bar2) begin
//last two notes were same duration etc.
                                          //n2 will be a "right" type
                                          n2_sprite = (3* p_nduration +
2 + memory_bar_d) ;
                                          //n1 will be a "middle" type
                                          n1_sprite = (3*
n2_nduration_old + 1 + memory_bar_d);
                                          //n0 will be a "left" type
                                          n0_sprite = (3*
n1_nduration_old + memory_bar_d);
                                          end

                                          else begin //last two notes
were not the same duration etc.
                                          //n2 will be a "right" type
                                          n2_sprite = (3* p_nduration +
2 + memory_bar_d) ;
                                          //n1 will be a "left" type
                                          n1_sprite = (3*
n2_nduration_old + memory_bar_d);
                                          //n0 will be old n1
                                          n0_sprite = n1_sprite_old;
                                          end
                                    end
                              else begin //p type note - code the same
as for d expect use memory_bar_p instead
                                    if (bar1 & bar2) begin   //last
three notes were all same duration etc.
                                          //n2 will be a "right" type
                                          n2_sprite = (3* p_nduration +
2 + memory_bar_p) ;
                                          //n1 will be a "middle" type
                                          n1_sprite = (3*
n2_nduration_old + 1 + memory_bar_p);
                                          //n0 is also a "middle"type
                                          n0_sprite = (3*
n1_nduration_old + 1 + memory_bar_p);
                                          end
```

```
                                        else if (!bar1 & bar2) begin
//last two notes were same duration etc.
                                        //n2 will be a "right" type
                                        n2_sprite = (3* p_nduration +
2 + memory_bar_p) ;
                                        //n1 will be a "middle" type
                                        n1_sprite = (3*
n2_nduration_old + 1 + memory_bar_p);
                                        //n0 will be a "left" type
                                        n0_sprite = (3*
n1_nduration_old + memory_bar_p);
                                        end

                                        else begin //last two notes
were not the same duration etc.
                                        //n2 will be a "right" type
                                        n2_sprite = (3* p_nduration +
2 + memory_bar_p) ;
                                        //n1 will be a "left" type
                                        n1_sprite = (3*
n2_nduration_old + memory_bar_p);
                                        //n0 will be old n1
                                        n0_sprite = n1_sprite_old;
                                        end
                                  end

                              //Deal with dots, sharps and durations
                              n2_dot = p_dot;
                              n1_dot = n2_dot_old;
                              n0_dot = n1_dot_old;


                              n2_sharp = p_sharp;
                              n1_sharp = n2_sharp_old;
                              n0_sharp = n1_sharp_old;


                              n2_nduration = p_nduration;
                              n1_nduration = n2_nduration_old;
                              n0_nduration = n1_nduration_old;
                              end

default:                      begin
                              n2_sprite = memory_special+1;
                              n1_sprite = memory_special+1;
                              n0_sprite = memory_special+1;

                              //Deal with dots
                              n2_dot = 0;
                              n1_dot = 0;
                              n0_dot = 0;

                              //Deal with sharps
                              n2_sharp = 0;
                              n1_sharp = 0;
                              n0_sharp = 0;
```

```
                              n2_nduration = p_nduration;
                              n1_nduration = n2_nduration_old;
                              n0_nduration = n1_nduration_old;
                              end
endcase

//n_pitch and n_octave signals
case (controller_state)

controller_setup:       begin
                              //At setup all sprites are 'static'
(clefs and time signatures)
                              n2_pitch = static_pitch;
                              n1_pitch = static_pitch;
                              n0_pitch = static_pitch;

                              n2_octave = static_octave;
                              n1_octave = static_octave;
                              n0_octave = static_octave;
                              end

controller_drest:       begin
                              //When drawing a rest the rest sprite is
'static'
                              n2_pitch = p_npitch;
                              n1_pitch = static_pitch;
                              n0_pitch = n2_pitch_old;


                              n2_octave = p_octave;
                              n1_octave = static_pitch;
                              n0_octave = n2_octave_old;
                              end

default:                      begin
                              n2_pitch = p_npitch;
                              n1_pitch = n2_pitch_old;
                              n0_pitch = n1_pitch_old;

                              n2_octave = p_octave;
                              n1_octave = n2_octave_old;
                              n0_octave = n1_octave_old;
                              end
endcase

end

assign clear_screen = (slice_count == 120);

endmodule
```

Labkit.v

```
module videolabkit   (beep, audio_reset_b, ac97_sdata_out,
ac97_sdata_in, ac97_synch,
             ac97_bit_clock,

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

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

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

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

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

             clock_feedback_out, clock_feedback_in,


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

             rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

             mouse_clock, mouse_data, keyboard_clock, keyboard_data,

             clock_27mhz, clock1, clock2,

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

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

             switch,

             led,

             user1, user2, user3, user4,

             daughtercard,

             systemace_data, systemace_address, systemace_ce_b,
```

```
             systemace_we_b, systemace_oe_b, systemace_irq,
systemace_mpbrdy,

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

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

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

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

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

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

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

   input  clock_feedback_in;
   output clock_feedback_out;

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

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;
```

```
   input  clock_27mhz, clock1, clock2;

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

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

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

   inout [43:0] daughtercard;

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

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


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

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

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

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

   // Video Input
   assign tv_in_i2c_clock = 1'b0;
```

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

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

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

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

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

   /*
   // LED Displays
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
```

```
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
   // disp_data_in is an input
      */

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


////////////////////////////////////////////////////////////////////////
/////
   //
   // lab4 : a simple pong game
   //

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

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

```
   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;


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

   // UP, DOWN, LEFT and RIGHT buttons for user input
   wire up,down, left, right;
   debounce db2(reset, clock_65mhz, ~button_up, up);
   debounce db3(reset, clock_65mhz, ~button_down, down);
   debounce db4(reset, clock_65mhz, ~button_right, right);
   debounce db5(reset, clock_65mhz, ~button_left, left);

   wire nextnote;
   debounce db6(reset, clock_65mhz, ~button3, nextnote);


      ///////////////////////////////////////
      ///////////////////////////////////////
      // SO BEGINNETH ADAMS CODE
      ///////////////////////////////////////
      ///////////////////////////////////////

/*
      Copy procedure:
      -copy changed .v files into ./Source
      -copy changed xco and .v files from logiccore modules into ./
        -delete any changed .ngo files from ./
      -make sure everything is off read-only
      -regenerate cores (?)
      -compile
*/
      ///////////////////////////////////
      // COPY AFTER THIS LINE
      ///////////////////////////////////

      wire clk = clock_65mhz;

      // Creates debounced/synced switches (sswitch)

      debounce dbswitch4(reset,clk,switch[3],sswitch3);
      debounce dbswitch5(reset,clk,switch[4],sswitch4);
      // Creates debounced/synced buttons (sbuttons)
      wire sbutton0,sbutton1,sbutton2,sbutton3;
      debounce dbbutton0(reset,clk,~button0,sbutton0);
      debounce dbbutton1(reset,clk,~button1,sbutton1);
      debounce dbbutton2(reset,clk,~button2,sbutton2);
      debounce dbbutton3(reset,clk,~button3,sbutton3);
      debounce dbbutton_left(reset,clk,~button_left,sbutton_left);
      reg old_sbutton_left;
```

```
	reg sbutton_left_pulse;
	always @ (posedge clk) begin
		old_sbutton_left <= sbutton_left;
		sbutton_left_pulse <= ~old_sbutton_left & sbutton_left;
	end

	///////////////////////////////////////
   // WIRES
	///////////////////////////////////////


	wire khz_enable;
   wire [19:0] from_ac97_data, to_ac97_data, ac97_data_dds;
	wire [14:0] char_freq;
	wire clk_sample;  // Synchronous version of audio_ready
	wire tc_done;
	wire fft_unload;
	wire le;
	wire one_32nd;


	wire audio_ready;


	parameter fft_length = 8192;    // (fft_length-1)            1023
	parameter fft_index_bits = 13;  // (fft_index_bits-1)  9
	parameter fft_in_bits = 20;     // (fft_in_bits-1)           19
	parameter fft_out_bits = 34;    // (fft_out_bits-1)    30

	reg [(fft_in_bits-1):0] xn_re;                  // Signal input
into FFT
	wire [(fft_index_bits-1):0] readaddr;
	// Address to read from in ring buffer
	wire [(fft_index_bits-1):0] xk_index;
	wire [(fft_out_bits-1):0] xk_re;
	wire [(fft_out_bits-1):0] abs_xk_re = xk_re[(fft_out_bits-1)] ?
~xk_re[(fft_out_bits-1):0] + 1 : xk_re;
	wire [(fft_out_bits-1):0] xk_im;
	wire [(fft_out_bits-1):0] abs_xk_im = xk_im[(fft_out_bits-1)] ?
~xk_im[(fft_out_bits-1):0] + 1 : xk_im;

	wire [(fft_out_bits):0] xk_magsq = abs_xk_re + abs_xk_im; // Just
fake "magnitude"

	///////////////////////////////////////
   // 1KHZ DIVIDER
	///////////////////////////////////////

	divider onekhz(clk, reset, khz_enable);




	///////////////////////////////////////
   // 32ND OF A BEAT GENERATOR
```

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


      wire [7:0] bpm;
      one32ndgen one32ndgen1(clk, reset, bpm, one_32nd);


      reg [4:0] met_count;
      always @ (posedge clk) met_count <= one_32nd ? met_count + 1 :
met_count;
      wire metronome = (met_count == 0);





      ///////////////////////////////////////
   // AUDIO RING BUFFER
      ///////////////////////////////////////

      reg [(fft_in_bits-1):0]             audiodat[(fft_length - 1):0];
      // Audio ring buffer (1024 20-bit samples)
      reg [(fft_index_bits-1):0]    adptr = 0;
                  // Audio data pointer

      always @ (posedge clk) begin
            if (clk_sample) begin
                  // Inverts the top bit to store the data as unsigned
                  audiodat[adptr] <= {~to_ac97_data[(fft_in_bits-
1)],to_ac97_data[(fft_in_bits-2):0]};
                  adptr <= adptr + 1;
            end
      end





      ///////////////////////////////////////
   // DDS
      ///////////////////////////////////////


      wire [63:0] hex_dds;
      ddstester ddstest(clk, button_up, button_down, audio_ready,
reset, switch,
                  ac97_data_dds, clk_sample, hex_dds);


      assign to_ac97_data = ~sbutton3 ? from_ac97_data : ac97_data_dds;





      ///////////////////////////////////////
   // DFT
```

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

      // 32.5 MHz clock ce for the fft
      reg fftcount;
      wire clk_fft = (fftcount == 0);
      always @ (posedge clk) fftcount <= fftcount + 1;


      always @ (posedge clk) xn_re <= audiodat[readaddr];

      dft dft1(clk, clk_fft, khz_enable, adptr, xn_re, le, readaddr,
// GIVE ME A SHOT
                  fft_unload, xk_index, xk_re, xk_im);
      defparam dft1.fft_length = fft_length;                       //
(fft_length-1)                    1023
      defparam dft1.fft_index_bits = fft_index_bits;  //
(fft_index_bits-1)          9
      defparam dft1.fft_in_bits = fft_in_bits;               //
(fft_in_bits-1)             19
      defparam dft1.fft_out_bits = fft_out_bits;             //
(fft_out_bits-1)            30


      reg [63:0] hex_dft=0;
      reg [15:0] q1hex_dft=0;
      reg [15:0] q2hex_dft=0;
      reg [15:0] q3hex_dft=0;
      reg [15:0] q4hex_dft=0;
      always @ (posedge clk) begin
            q1hex_dft <= fft_unload ? q1hex_dft + 1 : q1hex_dft;
            q2hex_dft <= clk_fft ? q2hex_dft + 1 :
q2hex_dft;//(readaddr == 1000) ? q2hex_dft + 1 : q2hex_dft;
            q3hex_dft <= readaddr;// ? q3hex_dft + 1 : q3hex_dft;
            q4hex_dft <= (xk_index == 1000) ? q4hex_dft + 1 :
q4hex_dft;
            hex_dft <= {q4hex_dft,q3hex_dft,q2hex_dft,q1hex_dft};
      end




      ///////////////////////////////////////
   // TONE CONVERTER
      ///////////////////////////////////////


      wire [63:0] hex_tc;
      toneconv toneconv1(clk, clk_fft, switch[5],switch[4:0],
abs_xk_re, abs_xk_im, xk_index, fft_unload,
                  char_freq, tc_done, hex_tc);
      defparam toneconv1.fft_length = fft_length;
      // (fft_length-1)                  1023
      defparam toneconv1.fft_index_bits = fft_index_bits;         //
(fft_index_bits-1)          9
      defparam toneconv1.fft_in_bits = fft_in_bits;
      // (fft_in_bits-1)                 19
```

```
    defparam toneconv1.fft_out_bits = fft_out_bits;
    // (fft_out_bits-1)                 30




    ///////////////////////////////////////
  // TONE LUT
    ///////////////////////////////////////

    wire [3:0] octave;
    wire [3:0] note;
    wire [63:0] hex_lut;

    tonelut
tonelut1(clk,clk_fft,char_freq,tc_done,note,octave,hex_lut);





    ///////////////////////////////////////
  // SCORE CONVERTER
    ///////////////////////////////////////

    wire [11:0] startbeat;
    wire [3:0] score_octave;
    wire [3:0] score_note;
    wire [3:0] score_duration;
    wire [11:0] score_startbeat;
    wire score_ready;
    wire [63:0] hex_sc;
    wire [63:0] hex_sc2;

    scoreconv scoreconv1(clk,khz_enable,note,octave,one_32nd,

    startbeat,score_octave,score_note,score_duration,score_startbeat,
score_ready, hex_sc2);

    reg [15:0] q1hex_sc=0;
    reg [15:0] q2hex_sc=0;
    reg [15:0] q3hex_sc=0;
    reg [15:0] q4hex_sc=0;
    always @ (posedge clk) begin
          q1hex_sc <= score_octave;//khz_enable ? q1hex_sc + 1 :
q1hex_sc;
          q2hex_sc <= score_note;//one_32nd ? q4hex_sc + 1 :
q4hex_sc;
          q3hex_sc <= score_ready ? q3hex_sc + 1 : q3hex_sc;
          q4hex_sc <= score_startbeat;
    end
    assign hex_sc = {q1hex_sc,q2hex_sc,q3hex_sc,q4hex_sc};


                /*
```

```
	///////////////////////////////////////
   // PLAYBACK BUFFER
	///////////////////////////////////////

	reg [8:0] playback_index=0;
	reg [23:0] playback_buffer[511:0];
	always @ (posedge clk) begin
		if (score_ready & ~playing) begin
			playback_buffer[playback_index] <=
{score_octave,score_note,score_duration,score_startbeat};
			playback_index <= playback_index + 1;
		end
	end
			 */

	///////////////////////////////////////
   // AC97
	///////////////////////////////////////
	// Synchronous (but skewed) 65/2 = 32.5 MHz clock for the ac97
	reg clk_ac97=0;
	reg clkcountac97=0;
	always @ (posedge clk) begin
		clkcountac97 <= clkcountac97+1;
		clk_ac97 <= clkcountac97 == 0;
	end

   audio myaudio(clk_ac97, power_on_reset, from_ac97_data,
to_ac97_data,
		 audio_ready, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
		 ac97_synch, ac97_bit_clock);
   defparam myaudio.VOLUME = 4'd10;





//	assign led = q3hex_sc[7:0];
	///////////////////////////////////////
   // HEX DISPLAY
	///////////////////////////////////////


	wire [63:0] hex_input =       sbutton0 ? hex_dds :           //
	[switch | sinwave |    | freqselect]
						sbutton1 ? hex_dft :
			// [unloadcount | clk_fft | readaddr | xk_index =
1000]
						sbutton2 ? hex_tc  :
			// [max_index | max_magsq |    | char_freq]
						hex_sc;
				// [octave | note | score_ready |
score_duration]

	// Slow albeit synchronous 65/8 = 8.125 MHz clock for the hex
	reg clk_hex=0;
	reg [3:0] clkcount=0;
```

```
      reg [63:0] b_hex_input;
      reg [63:0] b_hex_input2;
      always @ (posedge clk) begin
            clkcount <= clkcount+1;
            clk_hex <= clkcount == 0;
            b_hex_input <= (clkcount == 0) ? hex_input : b_hex_input;
            b_hex_input2 <= (clkcount == 0) ? b_hex_input :
b_hex_input2;
      end


   display_16hex d1(reset, clk_hex, b_hex_input2,
                disp_blank, disp_clock, disp_rs, disp_ce_b,
                disp_reset_b, disp_data_out);





      ///////////////////////////////////////
      ///////////////////////////////////////
      // SO BEGINNETH JOHNS CODE
      ///////////////////////////////////////
      ///////////////////////////////////////





   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

   // feed XVGA signals to user's pong game
   wire [2:0] pixel;
   wire phsync,pvsync,pblank;
   wire [1:0] major_state;
   wire [2:0] minor_state;
   wire major_toggle;
   wire [10:0] slice_x;
   wire [9:0] slice_y;
   wire [31:0] slice_pixels;
   wire artist_start, artist_done, busy;

   wire [2:0] n0_octave, n1_octave, n2_octave, n_octave;
   wire [3:0] n0_pitch, n1_pitch, n2_pitch;
   wire [5:0] n0_sprite, n1_sprite, n2_sprite;
   wire [5:0] n_sprite;
   wire [11:0] n0_countslice, n1_countslice, n2_countslice;

   wire [11:0]p_startbeat = score_startbeat;
   wire [3:0] p_duration = (score_duration);
   wire [2:0] p_octave = score_octave[2:0];
```

```
   wire [3:0] p_pitch = score_note;
   wire p_ready = score_ready;
   wire graphics_ready;       //start drawing
   wire [3:0] n_pitch;
   wire [1:0] clef;
   assign clef = {switch[6],1'b0};
   wire [2:0] timesig = 0;
   wire display_ready;
   wire [9:0] start_row;
   wire clear_screen;

//Debug
   wire go_bar, bar1, bar2;

graphics_controller_simple a4(p_octave, p_pitch, p_startbeat,
p_duration,
                                    p_ready,
                                    n0_octave, n0_pitch, n0_sprite,
n0_countslice, n0_sharp, n0_dot,
                                    n1_octave, n1_pitch, n1_sprite,
n1_countslice, n1_sharp, n1_dot,
                                    n2_octave, n2_pitch, n2_sprite,
n2_countslice, n2_sharp, n2_dot,
                                    clef, timesig,

                                    graphics_ready, display_ready,
                                    clear_screen,
                                    //playback,
                                    //Debug I/O begins
                                    //controller_state,
                                    go_bar, bar1, bar2,
                                    //Debug I/O ends
                                    reset, clock_65mhz);

buffer_manager a3(                  hcount, vcount, hsync, vsync,
blank,
                                    pvsync, phsync, pblank, pixel,
                                    slice_x, slice_y, slice_pixels,
busy,
                                    clear_screen,
                                    reset, clock_65mhz
                                    );

artist_module_minor a2 (            artist_start, artist_done,
                                    start_row, n_pitch, n_octave,
n_sprite, n_sharp, n_dot,
                                    clef,
                                    slice_y, slice_pixels, busy,
                                    //sprite_address, sprite_data,
                                    //slice_pitch_pixel,
sprite_y_offset,
                                    //sprite_count,
                                    //initial_sprite_address,
                                    minor_state,
                                    clock_65mhz, reset);
```

```
artist_module_major a1(       n0_octave, n0_pitch, n0_sprite,
n0_countslice, n0_sharp, n0_dot,
                                    n1_octave, n1_pitch, n1_sprite,
n1_countslice, n1_sharp, n1_dot,
                                    n2_octave, n2_pitch, n2_sprite,
n2_countslice, n2_sharp, n2_dot,
                                    graphics_ready,
                                    display_ready,
                                    slice_x, start_row,
                                    n_pitch, n_octave, n_sprite,
n_sharp, n_dot,
                                    artist_start, artist_done,

                                    /*
                                    //Debug
                                    //remd1, remd2, dividend, divisor1,
divisor2, quot1, quot2,
                                    start_divide,
                                    //n_countslice, old_n_countslice,
                                    //divide_ready1, divide_ready2,
divide_ready_count,
                                    */
                                    major_state,
                                    major_toggle,

                                    clock_65mhz, reset);

   /*
   user_input u1(up, down, left, right, clef, timesig, p_pitch,
p_octave, p_duration,
                        disp_blank, disp_clock, disp_rs, disp_ce_b,
                        disp_reset_b, disp_data_out,
                        nextnote, p_ready,
                        clock_65mhz, reset);
   */

   // switch[1:0] selects which video generator to use:
   //  00: user's pong game
   //  01: 1 pixel outline of active video area (adjust screen
controls)
   //  10: color bars
   reg [2:0] rgb;
   reg b,hs,vs;
   always @(posedge clock_65mhz) begin
      /*if (switch[1:0] == 2'b01) begin
       // 1 pixel outline of visible area (white)
       hs <= hsync;
       vs <= vsync;
       b <= blank;
       rgb <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 7
: 0;
      end else if (switch[1:0] == 2'b10) begin
       // color bars
       hs <= hsync;
       vs <= vsync;
       b <= blank;
       rgb <= hcount[8:6];
```

```
      end else begin
         // default: musicalator
       */
       hs <= phsync;
       vs <= pvsync;
       b <= pblank;
       rgb <= pixel;
      //end
   end

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

   assign led = ~{metronome,major_toggle, go_bar, bar2,
bar1,down,reset};

      assign analyzer1_data = {p_octave, p_pitch, p_ready, major_state,
minor_state, artist_start, artist_done, busy};
      assign analyzer1_clock = clock_65mhz;

endmodule

                                              /*
module
playback(clk,clk_dds,char_freq,startbeat,tc_done,note,octave,hex_lut);
      input clk;
      input clk_dds;
      input [14:0] char_freq;
       input tc_done;
    output reg [3:0] note;
    output reg [3:0] octave;
       output [63:0] hex_lut;


endmodule
                                                 */
```