Musical Feet:
A Step-by-Step Approach to Music Generation

Rajeev Nayak
Harley Zhang

ABSTRACT

The goal of this project is to generate random music in real time as a response to the user's walking pace. The music follows Western classical chord progressions, and the tempo and tonality change based on the frequency characteristics of the user's pace. An electromechanical pedometer senses the user's footsteps. The resulting analog signal is converted to a digital signal and sent to digital processing modules. The tempo of the music is controlled by the frequency of the footsteps, and the tonality responds to the level of fluctuation in the frequency. A finite state machine generates a chord progression in real time using these inputs. Based on the chords, an algorithm generates notes for each instrument of a string quartet following Western classical voice leading techniques. String instrument samples are stored and accessed based on the generated pitches, producing digital audio data that is combined and converted to an analog output. The user will be able to hear a pleasing musical strain while walking or running, providing auditory feedback of their pace.

Table of Contents

List of Tables and Figures

1 Overview

The Musical Feet system generates string quartet music in real time based on input from the user's footsteps. The tempo of the music reflects the user's pace, and the tonality of the music is determined by how much the user's pace fluctuates. This is achieved through a series of digital modules, in addition to analog circuitry at the system inputs and outputs. Figure 1 shows the high-level block diagram of the entire system. These different components can be grouped into four different sections: the pedometer input processing modules, the music composition modules, the music synthesis and audio modules, and the video output modules.

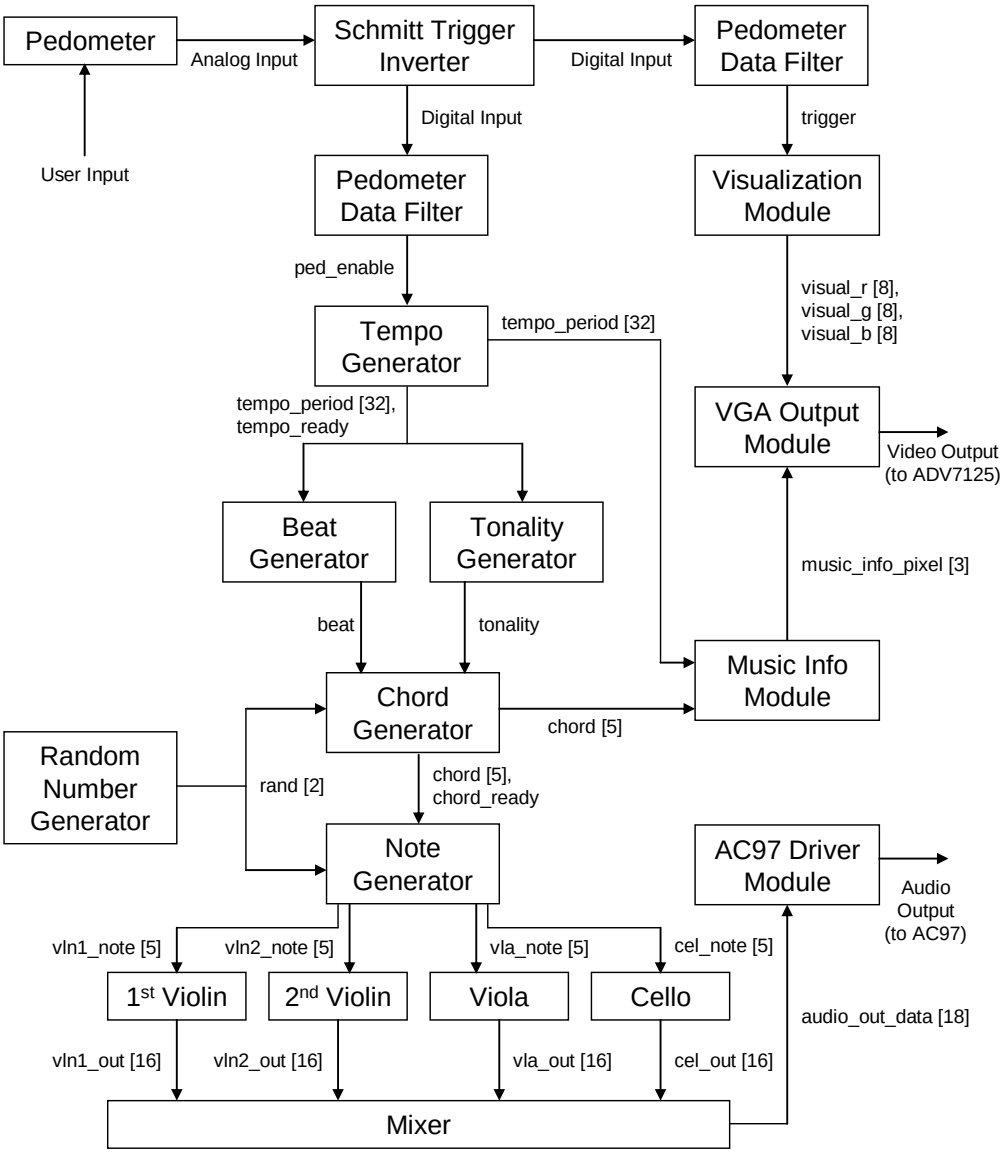

Figure 1. High level block diagram of Musical Feet system.
Numbers in brackets indicate bit width of signals.

The pedometer input processing modules take an analog signal produced by a pedometer whenever a footstep is taken and use it to generate a tempo and tonality for the music output. The analog signal from the pedometer is first converted to a digital signal which indicates when each footstep is taken. A tempo period is generated by the tempo generator module to match the interval between the user's successive footsteps. Then, based on these tempo periods, the tonality is determined by the tonality generator module. When the pace is fairly constant, the tonality will be major. Otherwise, the tonality will be minor. The tempo period is also used to produce a beat signal, which enables once every tempo period.

The music composition modules generate the notes for the string quartet based on the tonality and beat inputs from the input processing modules. On every beat, the chord generator module decides on a new chord using the previous chord and the tonality. It produces chord progressions that reflect the Western classical tradition, and it changes the key of the played music as the tonality changes. The note generator uses the chord to assign a note to each of the four instruments in a string quartet: a cello, a viola, and two violins.

The music synthesis and audio modules output the notes produced in the music composition modules as they would be played by a string quartet. Samples of each string of each instrument are stored in block RAMs. Oscillators access these samples at frequencies corresponding to the notes from the music composition modules. The resulting sample waveforms are modulated by an amplitude envelope, imitating the amplitude of a note bowed on a string. Once the amplitude is modulated, the signals from the four instruments are added together in the mixer and sent to the AC97 DAC. From there, the analog signal can be heard through speakers or headphones.

The video output modules display information related to the generated music and the system's input on a 1024x768 XVGA display. In Music Information mode, the video output shows the current tempo of the music in beats per minute and the current chord. In Visualization mode, the video output shows a footprint pattern each time a footstep is asserted. Thus, the video output modules show information from other parts of the Musical Feet system.

The modules of these four parts will be described in further technical detail in the following sections.

2 Pedometer Input Processing Modules

The Musical Feet system is controlled through the footsteps of the user. A pedometer worn by the user produces an analog signal each time the user takes a footstep, and this signal is then converted to a digital signal. From this footstep signal, the system generates a tempo and tonality for the music output. The tempo is then converted to a beat signal that determines when new notes will be played. All of the digital pedometer input processing modules are clocked off a 27MHz clock signal produced by the 6.111 Labkit.

2.1 Pedometer

The pedometer used in the Musical Feet system is Walking Advantage 342, by Sportline. Run off a 1.5V battery, it contains a mechanical arm that moves and induces a voltage whenever a step is taken. Wires are soldered onto the ground node and output node of the pedometer's PCB, and then connected onto the breadboard of the 6.111 Labkit for analog-to-digital conversion and filtering. The voltage at the output node is normally about 1.4V, and it drops briefly to about 0V when a step is taken.

2.2 Schmitt Trigger Inverter ADC

To convert the pedometer's analog signal to a digital signal, a Schmitt trigger inverter ADC (54LS14) is used. This chip is powered off a 5V source and compares the input value to its two internal thresholds. If the input is lower than the low threshold, the inverter outputs a high value of about 3.4V. If the input is higher than the high threshold, the inverter outputs a low value of about 0.25V. When the input is between the thresholds, the output maintains its previous value. Thus, the Schmitt trigger inverter asserts high when a step is taken and low otherwise. The output is noisy (Figure 2), so a filter is needed before it can be used by the digital system.

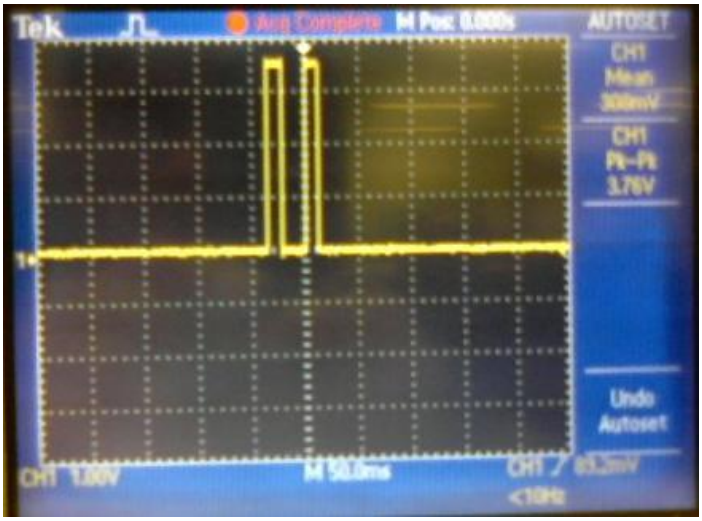

Figure 2. Output waveform of the Schmitt trigger inverter when a step is taken. The noisy analog signal produces a false assertion of the digital signal.

2.3 Pedometer Data Filter

The pedometer data filter removes false assertions of the digital signal produced by the ADC. When a rising edge occurs on this digital signal, the filter outputs a high value. For the next 0.2 seconds, the filter outputs a low value, even if there is a rising edge on the input. Otherwise, while the digital input is low, the output will also be low. This effectively eliminates false assertions because from observation, all false assertions occur within 0.2 seconds of the real assertion. After 0.2 seconds, the pedometer output is low until the next step is taken.

This also has the effect of limiting the speed at which the user can take steps to a rate of 300 steps per minute. For the tempo generation discussed below, this means that the maximum attainable tempo is 300 beats per minute (BPM). This is reasonable as an upper bound, since music generally is not written at higher tempos. Also, this allows sound synthesis effects like vibrato and enveloping to be clearly heard (Section 4.1.3).

2.4 Tempo Generator

The tempo generator takes the pedometer data filter's output and creates a 32-bit tempo period, measured in numbers of cycles from one beat to the next. The filter's output is high for a single cycle when each footstep is asserted and low otherwise. The tempo generator contains a 32-bit counter. Whenever a footstep is asserted, it stores the value of the count and resets the counter. The count is stored in a circular buffer with eight locations. The current buffer address is also incremented when a footstep is asserted. Since the buffer is circular, whenever a new footstep is taken, the new count overwrites the oldest count (Figure 3, p. 5). On the first footstep, every location in the buffer is initialized with the value of count, which corresponds to the number of cycles that have elapsed between the system start time and the footstep assertion.

To make tempo changes gradual for rapid changes in footstep speeds, the tempo generator takes a weighted average of the previous eight counts. The output tempo period is equal to ½ times the most recent count, plus ¼ times the next most recent count, and so on with the n^{th} most recent count weighted 2^{-n} up to n = 8. The tempo generator uses eight clock cycles to add these weighted counts, which are easily produced by bit-shifting to divide by appropriate powers of 2. After computing the tempo period, the tempo_ready signal is asserted for one cycle, signaling that the output tempo period is valid and ready for use in other modules.

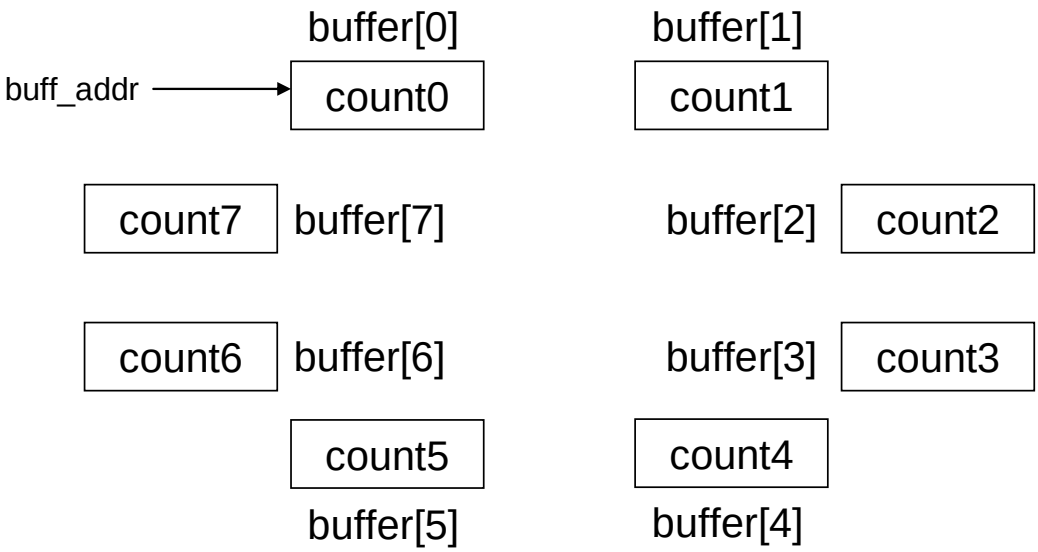

Figure 3(a). Circular buffer of tempo generator after eighth footstep. A new data value, count7, has just been placed in buffer[7], and the buffer address has been incremented to the location of the oldest data value.

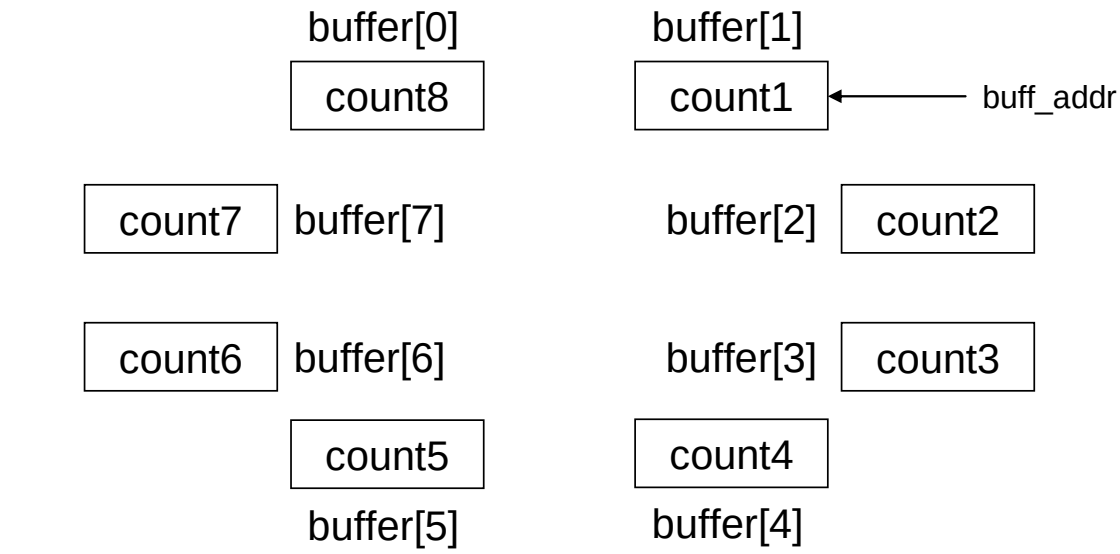

Figure 3(b). Circular buffer of tempo generator after ninth footstep. A new data value, count8, has just been placed in buffer[0], and the buffer address has been incremented to the location of the oldest data value.

2.5 Tonality Generator

The tonality generator uses the tempo period produced by the tempo generator to determine the tonality of the music output. The tonality output is a single bit, for which 0 corresponds to minor and 1 corresponds to major. The structure of the tonality generator is very similar to that of the tempo generator. Whenever the tempo_ready signal is asserted, the tonality generator takes the new tempo period and stores the absolute difference between the new tempo period and the previous tempo period. The differences are stored in a circular buffer with eight locations. The buffer address increments each time a new tempo is stored.

Like the tempo generator, the tonality generator takes a weighted average of the data in the buffer. An average value is generated equal to ½ times the most recent count, plus ¼ times the next most recent count, and so on with the n^{th} most recent count weighted 2^{-n} up to n = 8. This weighted average corresponds to the level of fluctuation in the speed of the user's footsteps. It is compared to a threshold value to produce the tonality output. If the average is less than the threshold, the user's footsteps are occurring at a fairly constant speed, and the tonality of the music is major. Otherwise, the user's footsteps fluctuate significantly, so the tonality is minor. The threshold value was empirically determined to maintain an appropriate balance between major and minor.

2.6 Beat Generator

The beat generator takes in the tempo period from the tempo generator when tempo_ready is asserted. Using a 32-bit counter, it produces a signal that asserts high for one cycle at a speed that matches the tempo period. The counter resets only when a beat is asserted. The tempo period is in number of cycles per beat, so when the value of the count matches the value of the tempo period, the output beat signal asserts high. If a new tempo period occurs on the input and is less than the current count, a beat is asserted and the counter resets and restarts counting. If the new tempo period is greater than the current count, then the counter continues counting and does not reset.

The initial value of the previous tempo period is set to 32'hFFFFFFFF, so if the system is turned on and no footsteps are ever taken, music will play with the corresponding tempo, which has a period of 32'hFFFFFFFF / 27MHz = 159 seconds. Thus, with no footsteps applied to the system input, the notes of the audio output still change, but only once per 159 seconds.

2.7 Testing of Pedometer Input Processing Modules

The tempo generator was tested by hooking up its input to a signal derived from a button on the 6.111 Labkit and its output to the Labkit's hexadecimal LED display. A module modified the debounced button signal to only assert high on the cycle that the button was pressed. When the button was pressed at a rate of once per second, the displayed output was close to 27000000, which is the number of cycles of the clock signal per second. Then, when the rate of button presses was suddenly changed to a constant faster speed, the displayed output gradually approached the number of cycles corresponding to the new speed. Also, the hex display showed the output of a counter that incremented whenever the tempo_ready signal was asserted. As expected, the counter incremented each time the button was pressed.

The tonality generator was wired to receive the outputs of the tempo generator. Its tonality output and also its weighted average of tempo differences were wired to the hex display. The button was pressed at varying intervals to check that the tonality value matched the comparison value of the weighted average and the threshold.

The beat generator was then wired to receive the outputs of the tempo generator. A counter that incremented whenever the beat signal asserted high was wired to the hex display. The counter was found to increment at a rate corresponding to various tempo periods.

The pedometer and analog circuitry were initially tested apart from the digital modules. The appropriate spots to solder wire onto the pedometer PCB were identified using a multimeter and oscilloscope. The wires were soldered in locations where they interfered least with the mechanical components of the pedometer. The pedometer was clipped onto various regions of the body to find where it detected steps most accurately. This was found to be at the front center of the user's pants, where it could pick up on the motion of both legs.

The ADC was then wired up and the pedometer signal was wired to its input. Originally, an operational amplifier had been used as a comparator to produce a digital signal based on a single threshold reference voltage. This was replaced by a Schmitt trigger inverter, which produces a less noisy output signal because it uses two thresholds. However, after examining the inverter's output on the oscilloscope, the digital signal was still found to be false asserting high less than 0.2 seconds after the initial assertion. A pedometer data filter was added, producing a clean output signal.

Finally, the input of the tempo generator module was changed from the Labkit button to the output of the pedometer data filter. Whenever a step was taken, the hex display showed that the outputs of the input processing modules were behaving as desired.

3 Music Composition Modules

The role of the music composition modules is to generate notes in real time for a string quartet based on the output of the pedometer input processing modules. The chord generator is responsible for figuring out the next chord that the quartet should play using the tonality it receives from the tonality generator. The note generator then figures out the next note that each instrument plays based on the new chord. The decisions made in the music composition had to involve some randomness, or else the music would have been entirely predictable and boring. In order to achieve this, the decisions made by the two composition modules take into account the value of a 2-bit number that comes from a random number generator. Altogether, these three modules handle all of the real-time music composition in the system.

3.1 Random Number Generator

The random number generator is implemented as a 10-bit Fibonacci linear feedback shift register (LFSR), with the low-order 2 bits as the output. Essentially, this is a 1023-state FSM, where each state corresponds to a distinct nonzero 10-bit number. The LFSR transitions to a new number at every clock cycle by shifting its previous number to the right by one bit. The new high-order bit is determined by two bits in the previous number called the taps. In a 10-bit LFSR, the optimal taps are bit 0 and bit 3. This means that at each cycle, the new high-order bit equals the XOR of the previous bits 0 and 3. This allows the LFSR to cycle through all 1023 nonzero values of the 10-bit number. The LFSR is depicted in Figure 4 below.

The LFSR value is initialized with a random seed generated by the random number generator module. A 10-bit counter in the module increments at each clock cycle, starting at the system reset. Every time the user reset is enabled, the value of the counter is used as a new seed for the LFSR. However, if the value of the seed is 0, the LFSR is seeded with 1 instead. This is required because if the LFSR is initialized with a value of 0, every subsequent value of the LFSR will also be 0. By assuring that the seed is not 0, the LFSR is guaranteed to enter its 1023-number cycle.

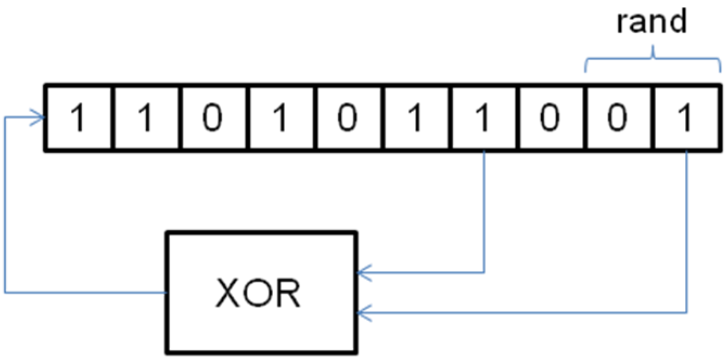

Figure 4. Diagram of random number generator. This is a 10-bit Fibonacci linear feedback shift register, and the two low-order bits are used as the "rand" output.

3.2 Chord Generator

The chord generator uses the tonality and random number outputs to determine the next chord based on the previous chord. The module is implemented as an FSM, where each state corresponds to a certain chord. There are six states: S_I, S_IV, S_V, S_i, S_iv, S_v. The first three of those states correspond to the major key tonic, subdominant, and dominant chords, respectively. Similarly, the last three correspond to the minor key tonic, subdominant, and dominant chords. The tonic note of the current key of the music is stored internally in the chord generator as a 4-bit number. The value of the note is encoded using values shown in Table 1 below. The key and the state of the FSM are used to determine the chord.

Table 1: Encoding of notes in an octave.

Value	Note
0	A
1	A#
2	B
3	C
4	C#
5	D
6	D#
7	E
8	F
9	F#
10	G
11	G#

The chord generator FSM transitions each time the beat output is enabled by the beat generator. The transitions from chord to chord are dictated by standard Western classical chord progressions. If the tonality of the music is major, the FSM will continue to transition between the S_I, S_IV, and S_V states, and the key will remain the same. If the tonality is minor, the FSM will transition between the S_i, S_iv, and S_v states, and the key will remain the same. However, if the tonality changes, the FSM will transition from a major key state to a minor key state or vice versa. During this transition, the value of the key may change. Some of the chord transitions allow the tonic note of the key to stay the same, with just the tonality of the key changing. However, other transitions only make sense when the key changes to a completely different tonic note. These transitions also conform to the Western classical tradition. The FSM transition table is shown on the next page in Table 2.

Table 2: State transition table of chord generator.

Current State	tonality	rand[0]	rand[1]	Next State	chord[3:0]
S_I	0	0	–	S_iv	key
S_I	0	1	0	S_i	key
S_I	0	1	1	S_v	key
S_I	1	0	–	S_I	key
S_I	1	1	0	S_IV	key
S_I	1	1	1	S_V	key
S_IV	0	0	–	S_i	key + 5
S_IV	0	1	–	S_v	key + 5
S_IV	1	0	–	S_V	key + 5
S_IV	1	1	0	S_IV	key + 5
S_IV	1	1	1	S_I	key + 5
S_V	0	0	–	S_i	key + 7
S_V	0	1	–	S_v	key + 7
S_V	1	0	–	S_V	key + 7
S_V	1	1	–	S_I	key + 7
S_i	0	0	–	S_i	key
S_i	0	1	0	S_iv	key
S_i	0	1	1	S_v	key
S_i	1	0	–	S_V	key
S_i	1	1	0	S_IV	key
S_i	1	1	1	S_I	key
S_iv	0	0	–	S_v	key + 5
S_iv	0	1	0	S_iv	key + 5
S_iv	0	1	1	S_i	key + 5
S_iv	1	0	–	S_V	key + 5
S_iv	1	1	–	S_I	key + 5
S_v	0	0	–	S_v	key + 7
S_v	0	1	–	S_i	key + 7
S_v	1	0	–	S_V	key + 7
S_v	1	1	–	S_I	key + 7

Once the FSM is done transitioning, the value of the chord is outputted. The high-order bit of the output indicates the tonality of the chord. This is set to 1 only if the current state is S_I, S_IV, S_V, or S_v, because they are all major chords. Otherwise, it is set to 0. The lower four bits of the output indicate the root note of the chord. This is easily calculated using the key and the state. The root note of a tonic chord is the tonic note of the key, so chord[3:0] equals the value of key when the state is either S_I or S_i. The root of a subdominant chord is 5 half steps above the tonic note, so chord[3:0] equals the value of key plus 5 when the state is either S_IV or S_iv. Finally, the root of a dominant chord is 7 half steps above the tonic note, so chord[3:0] equals the value of key plus 7 when the state is either S_V or S_v. When the new chord is calculated, the chord generator enables the chord_ready signal for one cycle and starts waiting for the next beat assertion.

3.3 Note Generator

The note generator calculates the notes that the instruments in the string quartet need to play based on the chord it receives from the chord generator. It makes sure that all of the notes of the chord are played so that the chord sounds full. It takes into consideration the previous notes played by the instruments as well, following standard Western classical voice leading techniques that make chord transitions sound smooth. It also adds some randomness in the choice of notes while maintaining these properties. It achieves this by searching for each instrument's note in succession before outputting all four notes. The note generator is implemented as an FSM, where each state involves a different stage of the note computation algorithm. The state transition diagram is depicted in Figure 5 below.

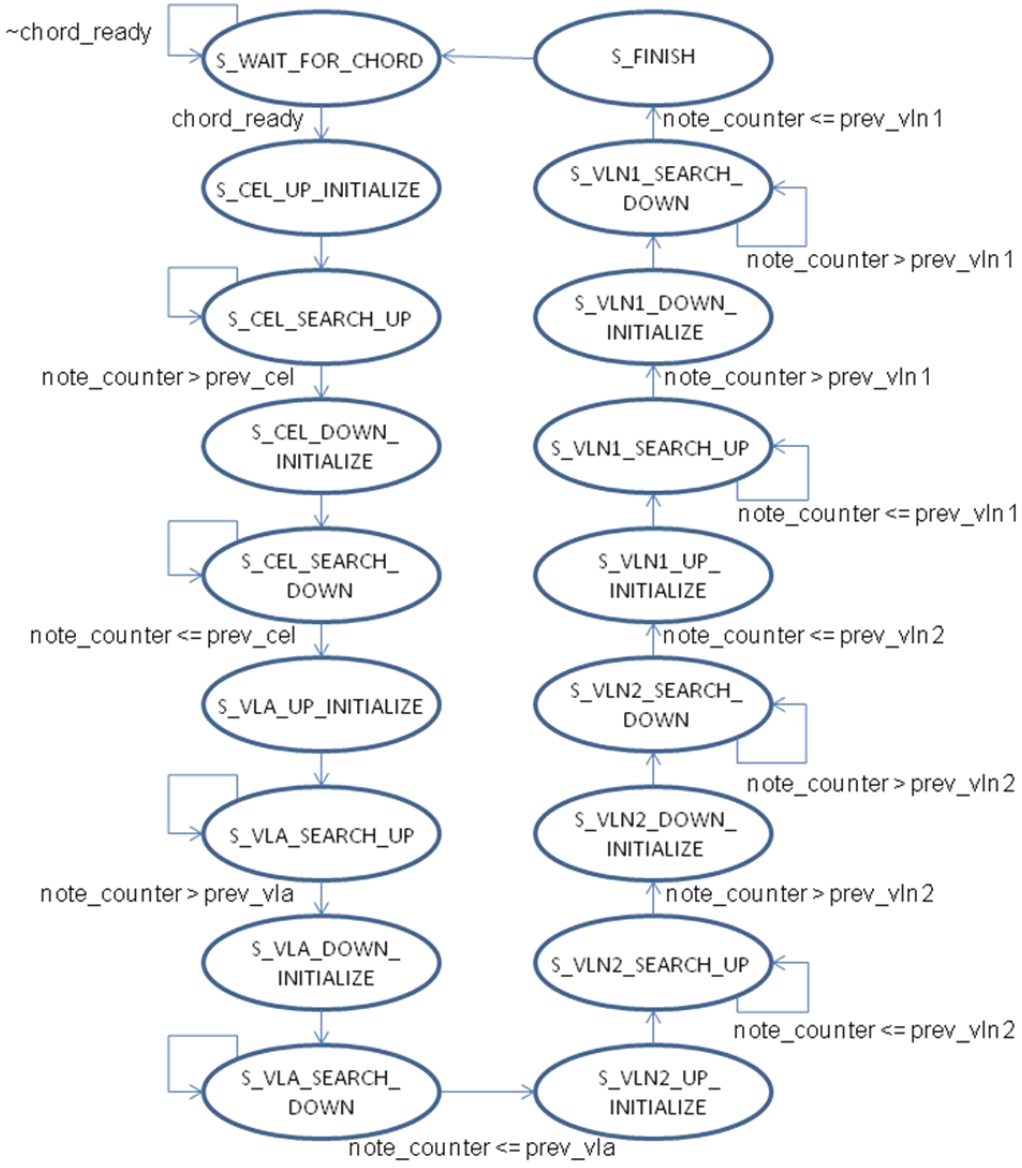

Figure 5. State transition diagram of note generator.

3.3.1 "Wait For Chord" State

The initial state of the FSM is the "wait for chord" state, in which the note generator remains idle, waiting for the next chord from the chord generator. When the chord_ready signal is enabled, the FSM transitions to the next state.

3.3.2 Bottom-Up Note Search

The next state initializes the note generator for the bottom-up cello note search. Each instrument's note search involves a bottom-up search and a top-down search. Each of these searches finds a number of candidate notes that could be assigned to the instrument. Once the search is over, the random number generator output is used to decide which of the candidates to assign to the instrument. The goal of the bottom-up cello note search is to find the nearest note lower than the previous cello note that fits into the new chord and meets an additional constraint. In order for a chord to sound full, the cello note has to play one of the bottom two notes in the chord: the root or the third. If the cello plays the top note in the chord, the fifth, the sound of the chord will be altered. Also, the bottom-up search checks to see if the previous cello note meets these criteria as well.

The bottom-up note search is performed using a 6-bit register called the note counter. The note counter stores the note that is currently being considered as a candidate in the note search. The note generator considers all of the notes from the bottom of the cello to the top of the violin, a range which spans multiple octaves. This means that the value of the note cannot be represented by the 4-bit encoding used in the chord generator. Instead, a 6-bit encoding is used, which is shown in Table 3 on the next page. The lowest note that a cello can play is C2 and the highest note that the violin is allowed to play in our system is C6. The encoding starts with A2 for simplicity, since the 4-bit octave encoding started with A.

In the cello bottom-up search, the note counter is initialized to the lowest note on the cello that fits in the current chord. The note counter increases after each clock cycle. This increase is determined by a minor FSM in the note generator module. This FSM keeps track of the type of note in the chord that the note counter is currently storing: the root, the third, or the fifth. If the note counter currently has the root of the chord, on the next clock cycle its value will be increased to the third of the chord. If it has the third of the chord, its value will be increased to the fifth of the chord. If it has the fifth of the chord, its value will be increased to the next root of the chord, which will be an octave above the previous root of the chord. For example, in an A major chord, all A's (A2, A3, A4, A5, and A6) are roots of the chord, all C#'s (C#2, C#3, C#4, and C#5) are thirds of the chord, and all E's (E2, E3, E4, and E5) are fifths of the chord.

Table 3: Encoding of notes for string quartet.

Value	Note	Value	Note
0	A2	**26**	B4
1	A#2	**27**	C4
2	B2	**28**	C#4
3	C2	**29**	D4
4	C#2	**30**	D#4
5	D2	**31**	E4
6	D#2	**32**	F4
7	E2	**33**	F#4
8	F2	**34**	G4
9	F#2	**35**	G#4
10	G2	**36**	A5
11	G#2	**37**	A#5
12	A3	**38**	B5
13	A#3	**39**	C5
14	B3	**40**	C#5
15	C3	**41**	D5
16	C#3	**42**	D#5
17	D3	**43**	E5
18	D#3	**44**	F5
19	E3	**45**	F#5
20	F3	**46**	G5
21	F#3	**47**	G#5
22	G3	**48**	A6
23	G#3	**49**	A#6
24	A4	**50**	B6
25	A#4	**51**	C6

As the note counter increments through notes in the chord, the bottom-up cello note search algorithm checks if they are either the root or the third of the chord. If a note fits the criteria, it is stored in a temporary register called lower_note. Also, the note type (root or third, in this case), is stored in a temporary register called lower_note_type. When the note counter exceeds the previous cello note value, the bottom-up search stops. This ensures that the value stored in the lower_note register is the nearest note lower than the previous note that fits the criteria. If the note counter also finds that the previous cello note fits the criteria, it stores the note type in a register called same_note_type.

3.3.3 Top-Down Note Search

Once these notes are found, the bottom-up search ends and the top-down search begins. Once again, the note counter is initialized with a note value, this time the highest possible note on the cello that fits into the current chord. Instead of increasing at each clock cycle, the note counter decreases during the top-down search. When a note fits the criteria, it is stored in the higher_note register and its type is stored in the higher_note_type register. After the note counter drops below the previous cello note, the top-down search is over, and the algorithm chooses the new cello note. During the two searches, the algorithm could have found anywhere between one and three candidates for the new cello note; some subset of the lower note, same note, and higher note. At the end of the top down search, the note generator module uses the random number generator output to select one of the found candidates to be the next cello note.

So far, the search has adhered to standard voice leading procedures, as the new note is guaranteed to be in close proximity to the previous note. However, the filling in of the chord has not been achieved yet. In order to do this, the module keeps track of which note types have been assigned at the end of each top-down search. In the viola, violin 2, and violin 1 note searches, the previous note types are taken into account. Just like the cello had the special criterion (it could not be the fifth of the chord), the successive note searches determine their criteria based on the previously chosen notes. By the end, two of the instruments should be playing the root of the chord, one should be playing the third, and one should be playing the fifth.

3.3.4 "Finish" State

The other three note searches proceed exactly as the cello note search, and once they are all done, the FSM transitions into the "finish" state for one clock cycle. In this state, it stores all of the note values so that they can be used as the previous values in the next note search. In addition, each of the note values is scaled by a certain value. This needs to happen because the modules in the audio synthesis section of the system use a different note indexing. Since each instrument is handled by a separate module in the synthesis section, the note encoding starts at the lowest note for each instrument. Therefore, the scaling is performed by subtracting the value of the lowest note for each instrument on the absolute scale defined in Table 3.

3.4 Testing of Music Composition Modules

The music composition modules were heavily tested using the 64-bit hexadecimal display on the 6.111 Labkit. The random number generator was tested quickly and easily by feeding the output to the display. The chord generator was also tested easily by outputting the state of the FSM and the current chord output onto the display while controlling the tonality with a switch and generating a beat every 3 seconds using a clock divider. However, difficulties arose while testing the complicated note generator module.

The note generator module behaved very erratically when it was first built. The hexadecimal display was also used to test this module by showing the value of each instrument's note in addition to the chord that came from the chord generator. The values of the notes were checked against the value of the chord to make sure that they corresponded. However, sometimes the notes behaved oddly. The instruments would often get stuck on the same four notes, no matter how much the chord changed. Also, the notes would sometimes cycle between a couple values, completely ignoring the changes in the chord.

After running into these problems over and over again, the decision was made to completely revamp the note generator. In fact, the current iteration of the note generator is drastically different from the version that was initially tested. In the old version, the note counter was never initialized more than once. It only performed a single bottom-up search, assigning the values of all four notes quickly. However, this made the logic extremely complicated and convoluted, which probably caused all of the errors. In the end, the decision was made to ignore the speed and efficiency of the module and focus on correctness and understandability. The note search logic is quite complicated, especially in a hardware implementation, so making the algorithm understandable is extremely important. Also, speed and efficiency were not as important as expected, because the extra 30 or so clock cycles that the new note generator takes to run are not noticeable to the human eye or ear. Once the note generator was remade from scratch, all of the erratic behavior was gone and it behaved perfectly, completing the music composition modules.

4 Music Synthesis and Audio Modules

The output audio of the Musical Feet system takes the form of a string quartet consisting of two violins, a viola, and a cello playing the notes produced by the Music Composition Modules. High-level modules for each instrument contain BRAMs to store sound samples, oscillators to select the correct pitches, and envelope generators to add sound effects to make the sounds more similar to real string instruments. The digital audio signals are then combined in a mixer and sent to an AC97 driver module. They are then converted into an analog signal through the AC97 DAC and sent to speakers or headphones to be played as sound. All of the music synthesis and audio modules are clocked off a 27MHz clock signal produced by the 6.111 Labkit.

4.1 Instrument Modules

The Musical Feet system contains three different instruments: the violin, viola, and cello. Each has its own high-level module that contains various submodules. The high-level modules differ from each other in the sound samples stored in their BRAMs. These three modules each take in a note from the note generator and output corresponding digital audio signals to the mixer. Figure 6 shows a block diagram of the violin module. The viola and cello modules have identical structures.

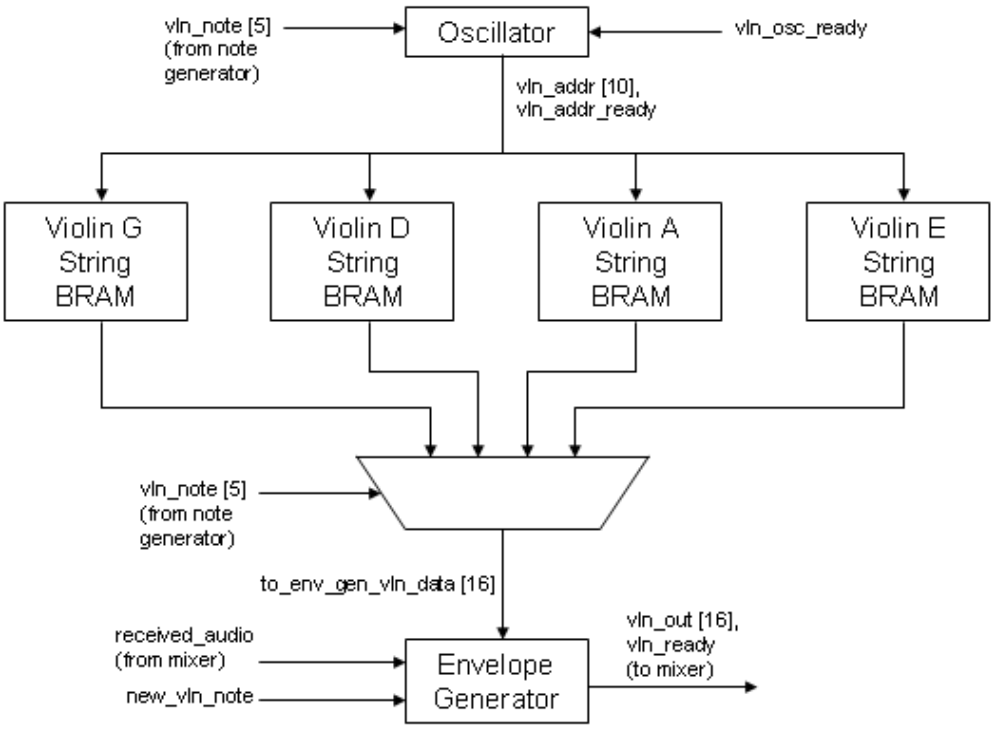

Figure 6. Block diagram of violin module. The viola and cello modules are identical to the violin module except they have their appropriate BRAMs and signal names.

4.1.1 String BRAMs

The string BRAMs each contain 16-bit wide samples of an open string played on a string instrument. Each high-level instrument module contains four different BRAMs, corresponding to the four open strings of each instrument. The samples were obtained from the Internet in the form of .aiff and .wav files (References, p. 27). The .aiff files were converted to .wav files. The .wav files were then processed in MATLAB to isolate single periods of the audio waveforms. The data from these single periods were then converted into .coe files that initialize the string BRAMs. There are a total of nine different BRAMs, since the viola's three higher strings have the same pitches as the violin's lower three strings. The viola and violin are similar enough in timbre that this sharing of strings is valid. The .wav files were sampled at 44.1kHz, so the high-level instrument modules get new sample data from the BRAMs every 612 cycles of the 27MHz clock (27MHz / 44.1kHz = 612). The appropriate BRAM is selected by the high-level instrument modules based on the selected note and the output data of the BRAM is then sent to the envelope generator module.

4.1.2 Oscillator

The oscillator determines the rate at which the address of the BRAMs should be incremented, thereby controlling the pitch of the output audio waveform. Since there are twelve half-steps in each octave for Western music, the frequency of any note is $2^{(1/12)}$ times the frequency of the note that is one half-step lower. Based on the note given by the note generator, the oscillator picks the appropriate power of $2^{(1/12)}$, accurate to ten binary decimal places, as the increment interval. On the next cycle, the interval is added to the internal address value. The actual address is outputted as the whole number part of the internal address value. Thus, the BRAM is accessed at the frequency that will produce audio data matching the input note's pitch.

Additional logic in the oscillator makes the address loop back to the beginning of the BRAM when the corresponding BRAM's maximum depth is reached. The oscillator also has an addr_ready signal that asserts when the oscillator has the next BRAM address.

4.1.3 Envelope Generator

The envelope generator reads each sample from the BRAM and modulates its amplitude in order to make the output waveform sound like a bowed string. It achieves this by applying an ADSR (attack, decay, sustain, and release) envelope on the BRAM sample waveform. When a note is initially played with a bow, the bow hits the string and the amplitude of the note increases rapidly. This is called the attack phase. Immediately after the attack, the amplitude of the note quickly decays to a steady amplitude. As the bow continues to run across the string, this amplitude is held fairly constant. This is called the sustain phase. Finally, when the bow is taken off the string, the note releases and the amplitude falls back down to zero. The envelope generator multiplies the BRAM samples by different values, creating this envelope. However, there is no release phase, since the four instruments never stop playing in the Musical Feet System. A graph of the amplitude envelope applied to two consecutive notes is shown in Figure 7 below.

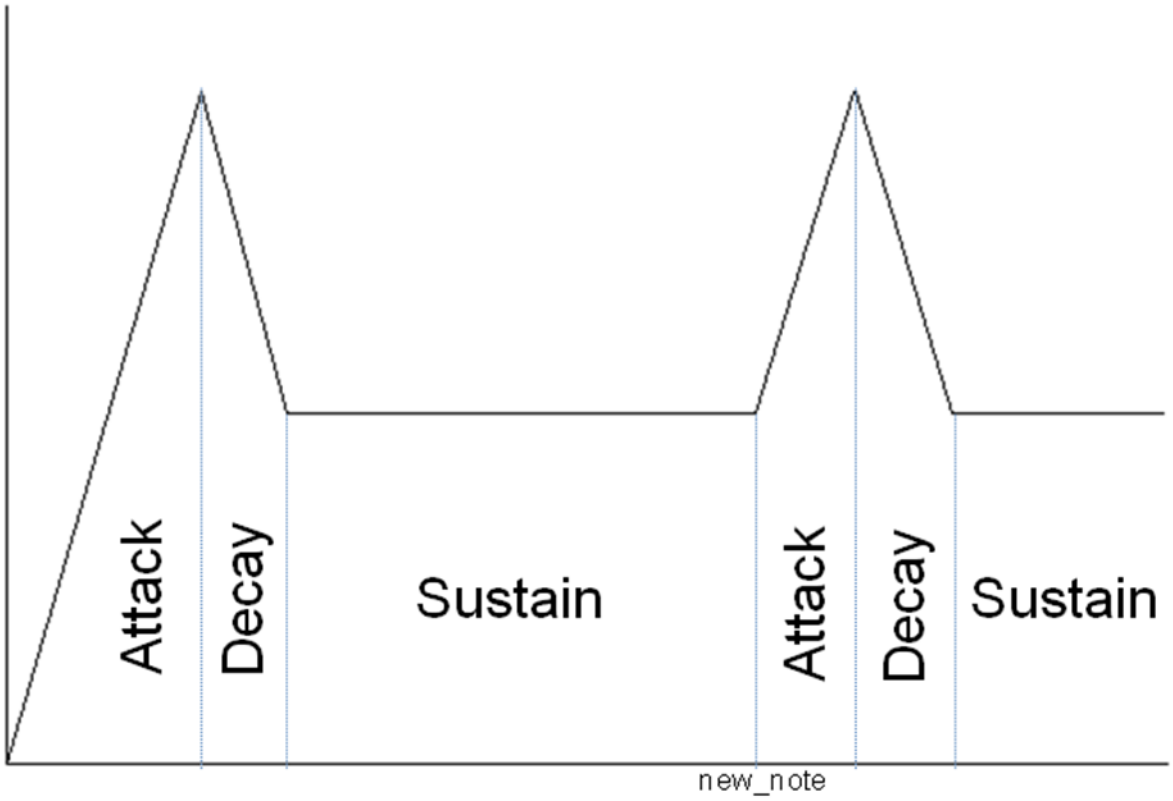

Figure 7. Amplitude envelope of two consecutive notes.

The envelope generator module is implemented using two FSM's. One controls the top-level behavior of the module, and the other keeps track of the envelope state: attack, decay, or sustain. The states of the top-level FSM will be discussed in detail.

4.1.3.1 "Wait For Sample" State

The initial state of the FSM is the "wait for sample" state. In this state, the envelope generator waits for the oscillator to assert its ready signal. When the oscillator indicates that the new address is ready, the envelope generator reads the next sample out of the BRAM. It also checks the new_note output of the instrument module. This indicates whether the new sample corresponds to a new pitch. If so, the envelope returns to the attack phase in order to attack the new note. After this, the FSM transitions to the next state.

4.1.3.2 "Adjust Envelope" State

In the "adjust envelope" state, the envelope generator decides whether it needs to transition to a new envelope state. During the attack and decay phases, the module uses a counter to keep track of how much time has passed. These two phases both have fixed lengths, so the module checks if the timer has reached the appropriate length. If so, the envelope state transitions, either from attack to decay or from decay to sustain. If the envelope is in the sustain state, it will stay there until the new_note signal is asserted. After the transition is decided, the FSM transitions again.

4.1.3.3 "Apply Envelope" State

In the "apply envelope" state, the sample is actually scaled to the appropriate value. Depending on the envelope state, the scaling is calculated differently. In the attack state, the sample is multiplied by the value of the envelope timer and shifted right by 15 bits. Since the largest possible value of the envelope timer is 2^{15}, the largest possible value of this scaling is 1. In the decay state, the sample is multiplied by the attack height (2^{15}) minus the value of the timer and then shifted by 15 bits. After the scaling is done in both of these cases, the envelope timer is incremented.

The sustain state does not utilize the envelope timer, but it uses its own timer to add vibrato to the envelope. Vibrato is created when a string instrument player rolls his finger back and forth across the string, making both the frequency and amplitude of the pitch rapidly fluctuate. The vibrato applied in this module only causes the amplitude to fluctuate, and it is employed during the sustain phase. The sample is scaled by the attack height minus the decay height plus the value of the vibrato timer, and then it is shifted by 15 bits. After this, the vibrato timer is either incremented or decremented, depending on the vibrato direction. When the vibrato timer reaches its maximum value, the vibrato direction changes to down, and when the vibrato timer reaches 0, the direction changes to up. This causes the amplitude of the note to rapidly fluctuate up and down during the sustain phase. Note that this fluctuation is not depicted in Figure 7.

Once the sample is scaled, the FSM transitions to its final state.

4.1.3.4 "Wait For Mixer" State

At this point, the envelope generator has successfully scaled the sample, so it sends out a ready signal to the mixer, which combines the signals sent by each of the four envelope generators. Once the mixer receives all four signals, it sends a received_audio signal back to each envelope generator. When the envelope generator receives this signal, it transitions back to the "wait for sample" state.

4.2 Mixer

The mixer module receives the scaled instrument samples from the four envelope generators and combines them into one signal to send to the AC97 DAC. It does this by simply adding together the four samples it receives. Once it receives ready signals from all four envelope generators, it starts adding up the samples. Since each sample is 16 bits wide, the mixer takes one clock cycle to add each one. If all four were added in the same clock cycle, timing constraints would be violated. Once all four signals are added, the mixer sends the received_audio signal to each envelope generator and sends the combined 18-bit signal to the AC97 DAC.

4.3 AC97 Driver

The AC97 driver converts the 18-bit digital audio signal sent by the mixer into an analog signal that can be outputted through speakers or headphones. The module is extremely similar to the 8-bit AC97 driver written by the 6.111 staff. There are only two minor changes. First, the AC97 input handling has been removed, since the Musical Feet system only has to provide the output. Second, the 8-bit signal that was padded with 12 zeros and set as the audio_out_data is now an 18-bit signal padded with 2 zeros. The same audio_out_data is still sent to both the left and right speakers.

4.4 Testing of Music Synthesis and Audio Modules

The testing of these modules was more difficult than other modules because they were all closely interdependent. They were first written and wired together, and then tested as a unit. There were two problems that became apparent during this testing.

At first, sound only came out when fewer than four instruments were sending their output signals into the mixer. This was caused by a timing constraint violation in the mixer, as there was not enough time in a single clock cycle to add four 16-bit numbers together. To fix this, the addition was split up into four clock cycles, adding one number per cycle. Afterwards, all four instruments could be heard at the same time.

Once all four pitches could be heard, it was apparent that the pitches deviated from the correct frequencies by a noticeable amount. This caused imperfect blending among the four instruments. To solve this problem, the oscillator intervals were extended to have more binary decimal places, resulting in a more accurate sampling frequency of the BRAMs. After this change, the music synthesis and audio modules behaved as expected.

5 Video Output Modules

The video output of the Musical Feet system has two modes: the Music Information mode and the Visualization mode. The mode is set by a switch on the 6.111 Labkit, which determines whether the display pixel is taken from the Music Information module or the Visualization module. In the Music Information mode, the display shows the current tempo of the output music in beats per minute (BPM), and the music's current chord. In the Visualization mode, a footprint image appears and fades away each time a footstep is taken by the user, allowing the user to observe whether the system has registered his steps.

The output is shown on a 1024x768 XVGA display, which requires a 65MHz clock for a 60Hz refresh rate. Thus, all of the video output modules are clocked off a 65MHz clock signal produced by the 6.111 Labkit. Signals coming from modules clocked off the 27MHz signal are synchronized to the 65MHz clock before being used, and the digital pedometer signal from the Schmitt trigger inverter is filtered for 0.2 seconds off the 65MHz clock (Section 2.2, p. 3).

5.1 XVGA Module

The XVGA module was written by the 6.111 staff. It generates the necessary horizontal and vertical sync signals, using counters to keep track of the horizontal and vertical coordinates of the current pixel. The sync signals are sent to the ADV7125 video DAC, while the count signals are used by the Music Information and Visualization modules.

5.2 Music Information Modules

The Music Information module takes synchronized data from the tempo generator and chord generator to display them on a monitor screen. It sends an output pixel value to the XVGA module based on the location of the current pixel. To produce the appropriate text, the Music Information module uses several submodules (Figure 7, p. 23).

5.2.1 String Display Module

The string display module was written by the 6.111 staff. Given horizontal and vertical coordinates, it produces the appropriate pixel values to display text input in the form of ASCII strings. Characters are produced based on a font ROM, which was written by Xilinx. The Music Information module uses four instances of the string display module to display "TEMPO: ", the tempo in beats per minute (BPM), "CHORD: ", and the key of the chord.

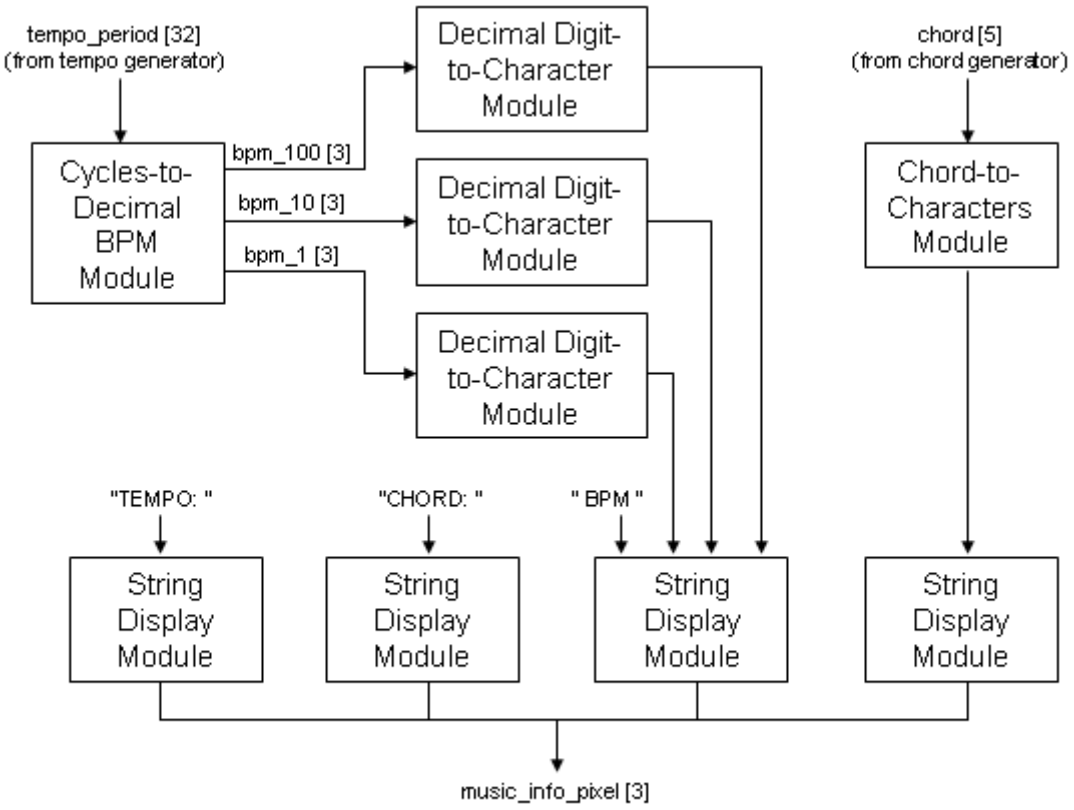

Figure 8. Block diagram of music information module.

5.2.2 Cycles-to-Decimal BPM Converter

The tempo period produced by the tempo generator is in number of cycles of a 27MHz clock per beat. For the Music Information display, this number needs to be converted into beats per minute. Moreover, the number should be displayed as a decimal number. The cycles-to-decimal BPM converter achieves this by dividing the number of cycles of a 27MHz clock in one minute (1.62e9) by the tempo period. The division is carried out by continuously subtracting the tempo period from 1.62e9 until subtraction would yield a negative number. The quotient is the number of subtractions that were performed. This number is the tempo in beats per minute.

To convert this tempo to a decimal number, the converter first continuously subtracts 100 until the result is less than 100. The number of subtractions performed is equal to the hundreds digit of the decimal number. Similarly, 10 is then continuously subtracted, and then 1, to get the tens digit and the ones digit of the decimal number. Since the tempo is limited to 300 BPM by the pedometer data filter (Section 2.3, p. 4), three decimal digits will suffice for any valid tempo period.

5.2.3 Decimal Digit-to-Character Converter

This module takes a decimal digit produced by the cycles-to-decimal BPM converter and outputs the corresponding 8-bit ASCII value, to be used in the string display module. For the hundreds digit, a value of 0 will produce a " " on the display, while a value of 0 for the tens or ones digit produce a "0" on the display.

5.2.4 Chord-to-Characters Converter

This module takes the 5-bit chord output from the chord generator and outputs the corresponding ASCII value of the string, to be used in the string display module. The high order bit of the chord signal indicates the tonality, while the four low order bits encode the root note of the chord.

5.3 Visualization Modules

The Visualization module contains two submodules, one that produces a left footprint and one that produces a right footprint. The module alternates between selecting the left and right footprint image pixels to send to the video output. Every time that the filtered pedometer signal asserts high, a white footprint is displayed on a black background and fades away. The location of the footprints on the screen is controlled by the Visualization module. They stay in the same horizontal positions, but their vertical positions are changed after each footstep, creating the image of feet walking forward and looping back to the bottom of the screen after they reach the top.

The Visualization module also has a 5-bit count signal that resets to 0 when a footstep is taken, increments when the vertical sync signal is asserted low, and stays at 31 when it is reached until the next footstep. This count is used by the footprint modules to fade the image of the footprints.

5.3.1 Footprint Modules

The left and right footprint modules are identical except that they contain BRAMs initialized with images of a left footprint and a right footprint, respectively. Each module keeps an address to its BRAM. The location of the image on the screen is given by the Visualization module. When the current pixel lies within the image's space, the footprint module outputs a pixel value based on the value at the BRAM's current address and increments the address. When the BRAM outputs a 0, the output pixel is black. When the BRAM output a 1, the output pixel is a shade of gray. The shade of gray is determined by setting the red, green, and blue pixel values to 248 minus the value of the count from the Visualization module times eight. Since the count ranges from 0 to 31, the pixel values range from 248 (nearly white) to 0 (black). Thus, the image fades completely after 31 low assertions of the vertical sync signal, which is about half a second.

5.4 Testing of Video Output Modules

The XVGA module and the other modules written by the 6.111 staff have been previously tested. The Music Information modules were tested by hardwiring numbers to the inputs of the converter modules and checking to see if the expected text was displayed on the screen. Then, the tempo generator and chord generator modules were hooked up to the inputs of the video modules, and the input of the tempo generator was hooked up to a Labkit button as described in Section 2.7. The values of the chord and tempo period signals were shown on the hex display and checked with the video output on the monitor to see if they matched. To test the Visualization modules, the signal that triggered footprints to appear was wired to a Labkit button. The expected pattern of footprints appeared.

6 Conclusion

The Musical Feet system differs from previous music generation systems in two innovative ways. First, the user has some degree of control over the music output through the speed of his footsteps, which allows for interesting possibilities. The system can provide auditory feedback of the user's walking and running paces, which could be useful for recognizing fatigue during athletic training. Also, it provides an entertaining form of exercise. Once the entire system was put together, it was used by several subjects, all of whom greatly enjoyed the novel experience.

Second, the Musical Feet system is unique in that it improvises its own music in real time instead of playing recorded samples. Not only does this significantly save memory, it also provides an interesting experience for the user. Adding randomness to the improvisation also prevents the user from hearing the same music each time he uses the system.

As described in previous sections, each part of the system was thoroughly tested for full functionality. Once the entire system was put together, further tests were performed. The audio and video outputs behaved as expected based on the user footstep input, and the music composition produced aurally pleasing music. During the course of this testing, one problem that did occur was in the initialization of the block RAMs containing the string samples. Occasionally, some of the block RAMs would be initialized with noisy samples, which resulted in a static-filled or distorted output sound. This problem could usually be resolved by restarting the 6.111 Labkit and reloading the Verilog code onto the FPGA.

The Musical Feet system has great potential for further improvement. The video output modules can be expanded to produce significantly flashier visualizations. The string sample BRAMs could contain longer samples so that the low frequency components of the sound waveforms are not lost. In addition, there could be direct control of the music volume based on the speed of the user's footsteps. Many parameters in the system can be tweaked to conform to each user's preferences, such as the tonality threshold and the envelope amplitude parameters. More advanced musical techniques can be replicated, such as different note articulations being imposed by the envelope generators. In an exercise setting, it may be helpful to have the music tempo slightly exceed the user's pace, encouraging the user to run faster. Another useful improvement for an exercise setting would be to provide a wireless interface between the pedometer and the 6.111 Labkit. Right now, the wires attached to the pedometer limit the user to walking or running in place, while a wireless interface would give users free rein to walk or run around while using the system.

7 References

String samples:

- University of Iowa Musical Instrument Samples:
  http://theremin.music.uiowa.edu/MIS.html

- Fitchsound Free Cello Samples
  http://fitchsounds.com/freestuff.html

Audio synthesis:

- MATLAB Help Desk:
  http://www.mathworks.com/access/helpdesk/help/toolbox/filterdesign/ref/

- The Amateur Gentleman's Introduction to the Principles of Music Synthesis:
  http://beausievers.com/synth/synthbasics/

- Articulation and Vibrato on the Violin:
  http://www.phys.unsw.edu.au/jw/violinarticulation.html

8 Appendix A: Verilog – Pedometer Input Processing Modules

8.1 Pedometer Data Filter

```
// ped_filter.v
// Author: Rajeev Nayak
// The ped_filter module filters the pedometer signal, ignoring all
// enables within .2 seconds of the first one.

module ped_filter #(parameter DELAY=5400000)   // .2 sec with a 27Mhz clock
                   (input reset,
                    input clock,
                    input noisy,
                    output reg clean);

   reg [23:0] count;
   reg waiting;

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

endmodule
```

8.2 Tempo Generator

```
// Tempo Generator Module
// Author: Harley Zhang
// Takes one-bit input from pedometer filter and calculates tempo periods

module tempo_gen (input reset,
                  input clock,
                  input ped_enable,
                  output reg tempo_ready,
                  output reg [31:0] tempo_period);

   wire [31:0] count;
   reg  [32:0] calc_period = 0;
   reg         buffer_reset = 1;
   reg         counter_reset = 0;
   reg         busy = 0;
   reg  [31:0] buffer[7:0];
   reg  [2:0]  buff_addr = 0;
   reg  [2:0]  addr_offset = 0;

   counter_32 counter(.reset(counter_reset),.clock(clock),.count(count));

   always @(posedge clock) begin

      if (reset) begin
         // System reset
         buffer_reset <= 1;
         counter_reset <= 1;
         calc_period <= 0;
         busy <= 0;
         buff_addr <= 0;
         addr_offset <= 0;
         tempo_ready <= 0;
      end

      else if (ped_enable) begin
      // First cycle of footstep
         busy <= 1;
         tempo_ready <= 0;
         counter_reset <= 1;
         buffer_reset <= 0;
         if (buffer_reset) begin
         // For first footstep, fill all buffer locations with the count
            buffer[0] <= count;
            buffer[1] <= count;
            buffer[2] <= count;
            buffer[3] <= count;
            buffer[4] <= count;
            buffer[5] <= count;
            buffer[6] <= count;
            buffer[7] <= count;
         end
         else begin
         // For subsequent footsteps, write over current buffer location
            buffer[buff_addr] <= count;
```

```
            buff_addr <= buff_addr + 1;
         end
      end

      else if (busy) begin
      // Use eight clock cycles to calculate tempo_period
         addr_offset <= addr_offset + 1;
         counter_reset <= 0;
         buffer_reset <= 0;
         if (addr_offset == 3'b111) begin
         // Finished calculating tempo_period
            busy <= 0;
            tempo_ready <= 1;
            tempo_period <= ((calc_period >> 1) + buffer[buff_addr +
                                                         addr_offset]) >> 1;
            calc_period <= 0;
         end
         else begin
            calc_period <= (calc_period >> 1) + buffer[buff_addr +
                                                       addr_offset];
            busy <= 1;
            tempo_ready <= 0;
         end
      end

      else begin
         tempo_ready <= 0;
         counter_reset <= 0;
         buffer_reset <= 0;
      end

   end

endmodule
```

8.3 Tonality Generator

```
// Tonality Generator Module
// Author: Harley Zhang
// Takes tempo_period from tonality generator module whenever
// it is ready, and then uses logic to determine tonality bit

module tonality_gen (input reset,
                     input clock,
                     input tempo_ready,
                     input [31:0] tempo_period,
                     output reg tonality);

   parameter threshold = 33'h40FFFF;

   reg [32:0] calc_tonality = 0;
   reg [31:0] prev_tempo_period;
   reg        buffer_reset = 1;
   reg        busy = 0;
   reg [31:0] buffer[7:0];
   reg [2:0]  buff_addr = 0;
   reg [2:0]  addr_offset = 0;

   always @(posedge clock) begin

      if (reset) begin
      // System reset
         buffer_reset <= 1;
         calc_tonality <= 0;
         busy <= 0;
         buff_addr <= 0;
         addr_offset <= 0;
      end

      else if (tempo_ready) begin
         busy <= 1;
         buffer_reset <= 0;
         prev_tempo_period <= tempo_period;
         if (buffer_reset) begin
            buffer[0] <= 0;
            buffer[1] <= 0;
            buffer[2] <= 0;
            buffer[3] <= 0;
            buffer[4] <= 0;
            buffer[5] <= 0;
            buffer[6] <= 0;
            buffer[7] <= 0;
         end
         else begin
         // For subsequent tempos, write over current buffer location
            buffer[buff_addr] <= (prev_tempo_period > tempo_period) ?
                                 prev_tempo_period - tempo_period :
                                 tempo_period - prev_tempo_period;
            buff_addr <= buff_addr + 1;
         end
      end
```

```
      else if (busy) begin
      // Use eight clock cycles to calculate calc_tonality
         addr_offset <= addr_offset + 1;
         buffer_reset <= 0;
         if (addr_offset == 3'b111) begin
         // Finished calculating calc_tonality
            tonality <= ((calc_tonality >> 1) +
                        buffer[buff_addr + addr_offset]) < threshold;
            // If the weighted average of the differences exceeds or equals
            // the threshold, tonality is 0, which is minor.  Otherwise, it
is
            // 1, which is major.
            calc_tonality <= 0;
            busy <= 0;
         end
         else begin
            calc_tonality <= (calc_tonality >> 1) +
                             buffer[buff_addr + addr_offset];
            busy <= 1;
         end
      end

      else begin
         buffer_reset <= 0;
      end

   end

endmodule
```

8.4 Beat Generator

```
// Beat Generator Module
// Author: Harley Zhang
// Takes tempo_period from tonality generator module
// and generates corresponding single-cycle enable signal

module beat_gen (input reset,
                 input clock,
                 input tempo_ready,
                 input [31:0] tempo_period,
                 output reg beat);

   reg counter_reset = 0;
   reg [31:0] prev_tempo_period = 32'hFFFFFFFF;
   wire [31:0] count;

   counter_32 counter(.reset(counter_reset),.clock(clock),.count(count));

   always @(posedge clock) begin

      if (reset) begin
         counter_reset <= 0;
         beat <= 0;
         prev_tempo_period <= 32'hFFFFFFFF;
      end

      else if (tempo_ready) begin
      // Read in new tempo period when ready
         prev_tempo_period <= tempo_period;
      end

      else if (count >= prev_tempo_period) begin
      // Assert output high and reset counter when desired count is
      // reached or surpassed
         counter_reset <= 1;
         beat <= 1;
      end

      else begin
         counter_reset <= 0;
         beat <= 0;
      end

   end

endmodule
```

9 Appendix B: Verilog – Music Composition Modules

9.1 Random Number Generator

```
// Author: Rajeev Nayak
// The random module generates a 2-bit pseudorandom number using a
// 10-bit Fibonacci linear feedback shift register.

module random(input clock,
              input reset,
              output [1:0] rand);

   reg [9:0] seed;
   reg [9:0] value;
   wire next;

   always @(posedge clock) begin
      // create a "random" seed by incrementing on every clock cycle
      // starting at system reset
      seed <= seed + 1;
      if(reset) begin
         // on the user reset, set the LFSR value to the current seed
         // value if the seed is 0, assign the value to 1
         if(seed == 0) value <= 1;
         else value <= seed;
      end
      else begin
         // shift the register
         value[0] <= value[1];
         value[1] <= value[2];
         value[2] <= value[3];
         value[3] <= value[4];
         value[4] <= value[5];
         value[5] <= value[6];
         value[6] <= value[7];
         value[7] <= value[8];
         value[8] <= value[9];
         value[9] <= next;
      end
   end

   // calculate the next value for the high-order bit using a Fibonacci
   // LFSR polynomial
   assign next = value[0] ^ value[3];
   // assign the output to be the low-order 2 bits of the LFSR value
   assign rand = value[1:0];

endmodule
```

9.2 Chord Generator

```
// chord_generator.v
// Author: Rajeev Nayak
// The chord_generator module uses an FSM to choose a chord
// based on the previous chord and the current tonality.

module chord_generator(input clock,
                       input reset,
                       input beat,
                       input tonality,
                       input [1:0] rand,
                       output [4:0] chord,
                       output reg chord_ready);

   parameter S_I = 0;  // major tonic chord
   parameter S_IV = 1; // major subdominant chord
   parameter S_V = 2;  // major dominant chord
   parameter S_i = 3;  // minor tonic chord
   parameter S_iv = 4; // minor subdominant chord
   parameter S_v = 5;  // minor dominant chord

   reg[3:0] key; // the current key of the music
   reg[2:0] state;

   always @(posedge clock) begin
      if(reset) begin
         key <= 0;
         state <= S_I;
         chord_ready <= 0;
      end
      // turn the chord_ready signal off after 1 cycle
      else if(chord_ready) chord_ready <= 0;
      // transition between chords on every beat
      else if(beat) begin
         case (state)

            S_I: begin
               state <= tonality ?
                        (rand[0] ? (rand[1] ? S_V : S_IV) : S_I) :
                        (rand[0] ? (rand[1] ? S_v : S_i) : S_iv);
               // key changes to iv on the S_I->S_i transition
               if(~tonality && rand[0] && ~rand[1]) begin
                  if(key >= 7) key <= key - 7;
                  else key <= key + 5;
               end
            end

            S_IV: begin
               state <= tonality ?
                        (rand[0] ? (rand[1] ? S_I : S_IV) : S_V) :
                        (rand[0] ? S_v : S_i);
               // key changes to vi on the S_IV->S_i transition
               if(~tonality && ~rand[0]) begin
                  if(key >= 3) key <= key - 3;
                  else key <= key + 9;
```

```
            end
         end

         S_V:  state <= tonality ?
                        (rand[0] ? S_I : S_V) :
                        (rand[0] ? S_v : S_i);

         S_i:  begin
            state <= tonality ?
                     (rand[0] ? (rand[1] ? S_I : S_IV) : S_V) :
                     (rand[0] ? (rand[1] ? S_v : S_iv) : S_i);
            // key changes to III on the S_i->S_I and S_i->S_IV
transitions
            if(tonality && rand[0]) begin
               if(key >= 9) key <= key - 9;
               else key <= key + 3;
            end
         end

         S_iv: state <= tonality ?
                        (rand[0] ? S_I : S_V) :
                        (rand[0] ? (rand[1] ? S_i : S_iv) : S_v);

         S_v:  state <= tonality ?
                        (rand[0] ? S_I : S_V) :
                        (rand[0] ? S_i : S_v);

         default: state <= S_I;

      endcase

      // assert that the new chord is ready
      chord_ready <= 1;
   end
end

// assign the tonality of the chord based on the state
assign chord[4] = state == S_I || state == S_IV ||
                  state == S_V || state == S_v;
// assign the root note of the chord based on the state and key
assign chord[3:0] = (state == S_I || state == S_i) ? key :
                    ((state == S_IV || state == S_iv) ?
                     ((key >= 7) ? key - 7 : key + 5) :
                     ((key >= 5) ? key - 5 : key + 7));

endmodule
```

9.3 Note Generator

```
// note_generator.v
// Author: Rajeev Nayak
// The note_generator module chooses four notes for a string quartet
// based on the chord produced by the chord_generator.

module note_generator(input clock,
                      input reset,
                      input chord_ready,
                      input [4:0] chord,
                      input [1:0] rand,
                                output [4:0] cel_note,
                      output [4:0] vla_note,
                      output [4:0] vln2_note,
                      output [4:0] vln1_note);

   // states for the note search
   parameter S_WAIT_FOR_CHORD = 0;
   parameter S_CEL_UP_INITIALIZE = 1;
   parameter S_CEL_SEARCH_UP = 2;
   parameter S_CEL_DOWN_INITIALIZE = 3;
   parameter S_CEL_SEARCH_DOWN = 4;
   parameter S_VLA_UP_INITIALIZE = 5;
   parameter S_VLA_SEARCH_UP = 6;
   parameter S_VLA_DOWN_INITIALIZE = 7;
   parameter S_VLA_SEARCH_DOWN = 8;
   parameter S_VLN2_UP_INITIALIZE = 9;
   parameter S_VLN2_SEARCH_UP = 10;
   parameter S_VLN2_DOWN_INITIALIZE = 11;
   parameter S_VLN2_SEARCH_DOWN = 12;
   parameter S_VLN1_UP_INITIALIZE = 13;
   parameter S_VLN1_SEARCH_UP = 14;
   parameter S_VLN1_DOWN_INITIALIZE = 15;
   parameter S_VLN1_SEARCH_DOWN = 16;
   parameter S_FINISH = 17;

   // note counter state
   parameter S_ROOT = 0;
   parameter S_THIRD = 1;
   parameter S_FIFTH = 2;

   // notes found in the previous search
   reg [5:0] prev_cel;
   reg [5:0] prev_vla;
   reg [5:0] prev_vln2;
   reg [5:0] prev_vln1;

   // notes in the current search
   reg [5:0] temp_cel;
   reg [5:0] temp_vla;
   reg [5:0] temp_vln2;
   reg [5:0] temp_vln1;

   // temporary registers for note search
   reg [5:0] lower_note; // stores the lower neighbor note in each
```

```
                     // instrument's search
reg [1:0] lower_note_type;
reg [1:0] same_note_type;
reg [5:0] higher_note; // stores the higher neighbor note in each
                       // instrument's search
reg [1:0] higher_note_type;
reg [1:0] last_note_type;
reg [2:0] note_found; // keeps track of which notes have been found in
                     // each instrument's search:
                     // 0=lower note, 1=same note, 2=higher note
reg [1:0] note_type_available[2:0]; // keeps track of which notes have
                                    // been assigned:
                                    // 0=root, 1=third, 2=fifth
reg [4:0] search_state;

// temporary registers for the note counter
reg [5:0] note_counter; // keeps track of the current note candidate
reg [1:0] counter_state;

always @(posedge clock) begin
   if(reset) begin
      temp_cel <= 9;
      temp_vla <= 13;
      temp_vln2 <= 9;
      temp_vln1 <= 14;
      prev_cel <= 12;
      prev_vla <= 28;
      prev_vln2 <= 31;
      prev_vln1 <= 36;
      search_state <= S_WAIT_FOR_CHORD;
      counter_state <= S_ROOT;
   end
   else begin
      case(search_state)

         // wait for the next chord to be chosen
         S_WAIT_FOR_CHORD: begin
            if(chord_ready) search_state <= S_CEL_UP_INITIALIZE;
         end

         // based on the chord, set the note counter to the
         // lowest possible note for the cello
         S_CEL_UP_INITIALIZE: begin
            // initialize the note counter
            // major chord
            if(chord[4]) case(chord[3:0])
               4'd0: begin
                  note_counter <= 6'd4;
                  counter_state <= S_THIRD;
               end
               4'd1: begin
                  note_counter <= 6'd5;
                  counter_state <= S_THIRD;
               end
               4'd2: begin
                  note_counter <= 6'd6;
                  counter_state <= S_THIRD;
```

```
      end
      4'd3: begin
         note_counter <= 6'd3;
         counter_state <= S_ROOT;
      end
      4'd4: begin
         note_counter <= 6'd4;
         counter_state <= S_ROOT;
      end
      4'd5: begin
         note_counter <= 6'd5;
         counter_state <= S_ROOT;
      end
      4'd6: begin
         note_counter <= 6'd6;
         counter_state <= S_ROOT;
      end
      4'd7: begin
         note_counter <= 6'd7;
         counter_state <= S_ROOT;
      end
      4'd8: begin
         note_counter <= 6'd8;
         counter_state <= S_ROOT;
      end
      4'd9: begin
         note_counter <= 6'd9;
         counter_state <= S_ROOT;
      end
      4'd10: begin
         note_counter <= 6'd10;
         counter_state <= S_ROOT;
      end
      4'd11: begin
         note_counter <= 6'd3;
         counter_state <= S_THIRD;
      end
      default: begin
         note_counter <= 6'd4;
         counter_state <= S_THIRD;
      end
   endcase
   // minor chord
   else case(chord[3:0])
      4'd0: begin
         note_counter <= 6'd3;
         counter_state <= S_THIRD;
      end
      4'd1: begin
         note_counter <= 6'd4;
         counter_state <= S_THIRD;
      end
      4'd2: begin
         note_counter <= 6'd5;
         counter_state <= S_THIRD;
      end
      4'd3: begin
```

```
            note_counter <= 6'd3;
            counter_state <= S_ROOT;
         end
         4'd4: begin
            note_counter <= 6'd4;
            counter_state <= S_ROOT;
         end
         4'd5: begin
            note_counter <= 6'd5;
            counter_state <= S_ROOT;
         end
         4'd6: begin
            note_counter <= 6'd6;
            counter_state <= S_ROOT;
         end
         4'd7: begin
            note_counter <= 6'd7;
            counter_state <= S_ROOT;
         end
         4'd8: begin
            note_counter <= 6'd8;
            counter_state <= S_ROOT;
         end
         4'd9: begin
            note_counter <= 6'd9;
            counter_state <= S_ROOT;
         end
         4'd10: begin
            note_counter <= 6'd10;
            counter_state <= S_ROOT;
         end
         4'd11: begin
            note_counter <= 6'd11;
            counter_state <= S_ROOT;
         end
         default: begin
            note_counter <= 6'd3;
            counter_state <= S_THIRD;
         end
      endcase

      // initialize search parameters
      note_found <= 3'b000;
      note_type_available[0] <= 2'b10; // allow 2 roots
      note_type_available[1] <= 2'b01; // allow 1 third
      note_type_available[2] <= 2'b01; // allow 1 fifth
      search_state <= S_CEL_SEARCH_UP;
   end

   // start incrementing the note counter, checking if the note
   // fits the following criteria:
   // 1. it has to be less than or equal to the previous cello note
   // 2. it can't be the fifth of the chord
   S_CEL_SEARCH_UP: begin
      // finding nearest lower neighbor
      if(note_counter < prev_cel) begin
         if(counter_state != S_FIFTH) begin
```

```
      lower_note <= note_counter;
      note_found[0] <= 1;
      lower_note_type <= counter_state;
      end
   end
   // previous note is still valid
   else if(note_counter == prev_cel) begin
      if(counter_state != S_FIFTH) begin
      note_found[1] <= 1;
      same_note_type <= counter_state;
      end
   end
   // counter went above previous note
   else search_state <= S_CEL_DOWN_INITIALIZE;
end

// the bottom-up search is done, so now set the note counter
// to the highest possible note for the cello
S_CEL_DOWN_INITIALIZE: begin
   // initialize the note counter
   // major chord
   if(chord[4]) case(chord[3:0])
      4'd0: begin
         note_counter <= 6'd28;
         counter_state <= S_THIRD;
      end
      4'd1: begin
         note_counter <= 6'd29;
         counter_state <= S_THIRD;
      end
      4'd2: begin
         note_counter <= 6'd30;
         counter_state <= S_THIRD;
      end
      4'd3: begin
         note_counter <= 6'd27;
         counter_state <= S_ROOT;
      end
      4'd4: begin
         note_counter <= 6'd28;
         counter_state <= S_ROOT;
      end
      4'd5: begin
         note_counter <= 6'd29;
         counter_state <= S_ROOT;
      end
      4'd6: begin
         note_counter <= 6'd30;
         counter_state <= S_ROOT;
      end
      4'd7: begin
         note_counter <= 6'd23;
         counter_state <= S_THIRD;
      end
      4'd8: begin
         note_counter <= 6'd24;
         counter_state <= S_THIRD;
```

```
      end
      4'd9: begin
         note_counter <= 6'd25;
         counter_state <= S_THIRD;
      end
      4'd10: begin
         note_counter <= 6'd26;
         counter_state <= S_THIRD;
      end
      4'd11: begin
         note_counter <= 6'd27;
         counter_state <= S_THIRD;
      end
      default: begin
         note_counter <= 6'd28;
         counter_state <= S_THIRD;
      end
   endcase
   // minor chord
   else case(chord[3:0])
      4'd0: begin
         note_counter <= 6'd27;
         counter_state <= S_THIRD;
      end
      4'd1: begin
         note_counter <= 6'd28;
         counter_state <= S_THIRD;
      end
      4'd2: begin
         note_counter <= 6'd29;
         counter_state <= S_THIRD;
      end
      4'd3: begin
         note_counter <= 6'd30;
         counter_state <= S_THIRD;
      end
      4'd4: begin
         note_counter <= 6'd28;
         counter_state <= S_ROOT;
      end
      4'd5: begin
         note_counter <= 6'd29;
         counter_state <= S_ROOT;
      end
      4'd6: begin
         note_counter <= 6'd30;
         counter_state <= S_ROOT;
      end
      4'd7: begin
         note_counter <= 6'd22;
         counter_state <= S_THIRD;
      end
      4'd8: begin
         note_counter <= 6'd23;
         counter_state <= S_THIRD;
      end
      4'd9: begin
```

```
            note_counter <= 6'd24;
            counter_state <= S_THIRD;
         end
         4'd10: begin
            note_counter <= 6'd25;
            counter_state <= S_THIRD;
         end
         4'd11: begin
            note_counter <= 6'd26;
            counter_state <= S_THIRD;
         end
         default: begin
            note_counter <= 6'd27;
            counter_state <= S_THIRD;
         end
      endcase

      search_state <= S_CEL_SEARCH_DOWN;
   end

   // start decrementing the note counter, checking if the note fits
   // the following criteria:
   // 1. it has to be greater than the previous cello note
   // 2. it can't be the fifth of the chord
   // once all notes greater than the previous note have been
   // considered, randomly choose the next note,
   // using candidates found in the bottom-up and top-down searches
   S_CEL_SEARCH_DOWN: begin
      // finding nearest upper neighbor
      if(note_counter > prev_cel) begin
         if(counter_state != S_FIFTH) begin
         higher_note <= note_counter;
         note_found[2] <= 1;
         higher_note_type <= counter_state;
         end
      end
      // counter went below previous note
      else begin
            case(note_found)
            3'b001: begin
               // assign lower note
               temp_cel <= lower_note;
               last_note_type <= lower_note_type;
            end

            3'b010: begin
               // assign same note
               temp_cel <= prev_cel;
               last_note_type <= same_note_type;
            end

            3'b100: begin
               // assign higher note
               temp_cel <= higher_note;
               last_note_type <= higher_note_type;
            end
```

```
3'b011: begin
   // assign lower note
   if(rand[0]) begin
      temp_cel <= lower_note;
      last_note_type <= lower_note_type;
   end
   // assign same note
   else begin
      temp_cel <= prev_cel;
      last_note_type <= same_note_type;
   end
end

3'b101: begin
   // assign lower note
   if(rand[0]) begin
      temp_cel <= lower_note;
      last_note_type <= lower_note_type;
   end
   // assign higher note
   else begin
      temp_cel <= higher_note;
      last_note_type <= higher_note_type;
   end
end

3'b110: begin
   // assign same note
   if(rand[0]) begin
      temp_cel <= prev_cel;
      last_note_type <= same_note_type;
   end
   // assign higher note
   else begin
      temp_cel <= higher_note;
      last_note_type <= higher_note_type;
   end
end

3'b111: begin
   if(rand[0]) begin
      // assign lower note
      if(rand[1]) begin
         temp_cel <= lower_note;
         last_note_type <= lower_note_type;
      end
      // assign higher note
      else begin
         temp_cel <= higher_note;
         last_note_type <= higher_note_type;
      end
   end
   // assign same note
   else begin
      temp_cel <= prev_cel;
      last_note_type <= same_note_type;
   end
```

```
         end

         default: begin
            // assign same note
            temp_cel <= prev_cel;
            last_note_type <= same_note_type;
         end
      endcase

      search_state <= S_VLA_UP_INITIALIZE;
   end
end

// based on the chord, set the note counter to the lowest
// possible note for the viola
S_VLA_UP_INITIALIZE: begin
   // initialize the note counter
   // major chord
   if(chord[4]) case(chord[3:0])
      4'd0: begin
         note_counter <= 6'd16;
         counter_state <= S_THIRD;
      end
      4'd1: begin
         note_counter <= 6'd17;
         counter_state <= S_THIRD;
      end
      4'd2: begin
         note_counter <= 6'd18;
         counter_state <= S_THIRD;
      end
      4'd3: begin
         note_counter <= 6'd15;
         counter_state <= S_ROOT;
      end
      4'd4: begin
         note_counter <= 6'd16;
         counter_state <= S_ROOT;
      end
      4'd5: begin
         note_counter <= 6'd17;
         counter_state <= S_ROOT;
      end
      4'd6: begin
         note_counter <= 6'd18;
         counter_state <= S_ROOT;
      end
      4'd7: begin
         note_counter <= 6'd19;
         counter_state <= S_ROOT;
      end
      4'd8: begin
         note_counter <= 6'd15;
         counter_state <= S_FIFTH;
      end
      4'd9: begin
         note_counter <= 6'd16;
```

```
            counter_state <= S_FIFTH;
        end
        4'd10: begin
            note_counter <= 6'd17;
            counter_state <= S_FIFTH;
        end
        4'd11: begin
            note_counter <= 6'd15;
            counter_state <= S_THIRD;
        end
        default: begin
            note_counter <= 6'd16;
            counter_state <= S_THIRD;
        end
    endcase
    // minor chord
    else case(chord[3:0])
        4'd0: begin
            note_counter <= 6'd15;
            counter_state <= S_THIRD;
        end
        4'd1: begin
            note_counter <= 6'd16;
            counter_state <= S_THIRD;
        end
        4'd2: begin
            note_counter <= 6'd17;
            counter_state <= S_THIRD;
        end
        4'd3: begin
            note_counter <= 6'd15;
            counter_state <= S_ROOT;
        end
        4'd4: begin
            note_counter <= 6'd16;
            counter_state <= S_ROOT;
        end
        4'd5: begin
            note_counter <= 6'd17;
            counter_state <= S_ROOT;
        end
        4'd6: begin
            note_counter <= 6'd18;
            counter_state <= S_ROOT;
        end
        4'd7: begin
            note_counter <= 6'd19;
            counter_state <= S_ROOT;
        end
        4'd8: begin
            note_counter <= 6'd15;
            counter_state <= S_FIFTH;
        end
        4'd9: begin
            note_counter <= 6'd16;
            counter_state <= S_FIFTH;
        end
```

```
      4'd10: begin
         note_counter <= 6'd17;
         counter_state <= S_FIFTH;
      end
      4'd11: begin
         note_counter <= 6'd18;
         counter_state <= S_FIFTH;
      end
      default: begin
         note_counter <= 6'd15;
         counter_state <= S_THIRD;
      end
   endcase

   // update availability and search parameters
   note_type_available[last_note_type] <=
      note_type_available[last_note_type] - 1;
   note_found <= 3'b000;
   search_state <= S_VLA_SEARCH_UP;
end

// start incrementing the note counter, checking if the note
// fits the following criteria:
// 1. it has to be less than or equal to the previous viola note
// 2. it has to fit into the chord based on the current note
//    availability
S_VLA_SEARCH_UP: begin
   // finding nearest lower neighbor
   if(note_counter < prev_vla) begin
      if(note_type_available[counter_state] > 0) begin
         lower_note <= note_counter;
         note_found[0] <= 1;
         lower_note_type <= counter_state;
      end
   end
   // previous note is still valid
   else if(note_counter == prev_vla) begin
      if(note_type_available[counter_state] > 0) begin
         note_found[1] <= 1;
         same_note_type <= counter_state;
      end
   end
   // counter went above previous note
   else search_state <= S_VLA_DOWN_INITIALIZE;
end

// the bottom-up search is done, so now set the note counter
// to the highest possible note for the viola
S_VLA_DOWN_INITIALIZE: begin
   // initialize the note counter
   // major chord
   if(chord[4]) case(chord[3:0])
      4'd0: begin
         note_counter <= 6'd36;
         counter_state <= S_ROOT;
      end
      4'd1: begin
```

```
            note_counter <= 6'd37;
            counter_state <= S_ROOT;
        end
        4'd2: begin
            note_counter <= 6'd33;
            counter_state <= S_FIFTH;
        end
        4'd3: begin
            note_counter <= 6'd34;
            counter_state <= S_FIFTH;
        end
        4'd4: begin
            note_counter <= 6'd35;
            counter_state <= S_FIFTH;
        end
        4'd5: begin
            note_counter <= 6'd36;
            counter_state <= S_FIFTH;
        end
        4'd6: begin
            note_counter <= 6'd37;
            counter_state <= S_FIFTH;
        end
        4'd7: begin
            note_counter <= 6'd35;
            counter_state <= S_THIRD;
        end
        4'd8: begin
            note_counter <= 6'd36;
            counter_state <= S_THIRD;
        end
        4'd9: begin
            note_counter <= 6'd37;
            counter_state <= S_THIRD;
        end
        4'd10: begin
            note_counter <= 6'd34;
            counter_state <= S_ROOT;
        end
        4'd11: begin
            note_counter <= 6'd35;
            counter_state <= S_ROOT;
        end
        default: begin
            note_counter <= 6'd36;
            counter_state <= S_ROOT;
        end
    endcase
    // minor chord
    else case(chord[3:0])
        4'd0: begin
            note_counter <= 6'd36;
            counter_state <= S_ROOT;
        end
        4'd1: begin
            note_counter <= 6'd37;
            counter_state <= S_ROOT;
```

```
      end
      4'd2: begin
         note_counter <= 6'd33;
         counter_state <= S_FIFTH;
      end
      4'd3: begin
         note_counter <= 6'd34;
         counter_state <= S_FIFTH;
      end
      4'd4: begin
         note_counter <= 6'd35;
         counter_state <= S_FIFTH;
      end
      4'd5: begin
         note_counter <= 6'd36;
         counter_state <= S_FIFTH;
      end
      4'd6: begin
         note_counter <= 6'd37;
         counter_state <= S_FIFTH;
      end
      4'd7: begin
         note_counter <= 6'd34;
         counter_state <= S_THIRD;
      end
      4'd8: begin
         note_counter <= 6'd35;
         counter_state <= S_THIRD;
      end
      4'd9: begin
         note_counter <= 6'd36;
         counter_state <= S_THIRD;
      end
      4'd10: begin
         note_counter <= 6'd37;
         counter_state <= S_THIRD;
      end
      4'd11: begin
         note_counter <= 6'd35;
         counter_state <= S_ROOT;
      end
      default: begin
         note_counter <= 6'd36;
         counter_state <= S_ROOT;
      end
   endcase

   search_state <= S_VLA_SEARCH_DOWN;
end

// start decrementing the note counter, checking if the note
// fits the following criteria:
// 1. it has to be greater than the previous viola note
// 2. it has to fit into the chord based on the current note
//    availability
// once all notes greater than the previous note have been
// considered, randomly choose the next note,
```

```
// using candidates found in the bottom-up and top-down searches
S_VLA_SEARCH_DOWN: begin
   // finding nearest upper neighbor
   if(note_counter > prev_vla) begin
      if(note_type_available[counter_state] > 0) begin
         higher_note <= note_counter;
         note_found[2] <= 1;
         higher_note_type <= counter_state;
      end
   end
   // counter went below previous note
   else begin
      case(note_found)
         3'b001: begin
            // assign lower note
            temp_vla <= lower_note;
            last_note_type <= lower_note_type;
         end

         3'b010: begin
            // assign same note
            temp_vla <= prev_vla;
            last_note_type <= same_note_type;
         end

         3'b100: begin
            // assign higher note
            temp_vla <= higher_note;
            last_note_type <= higher_note_type;
         end

         3'b011: begin
            // assign lower note
            if(rand[0]) begin
               temp_vla <= lower_note;
               last_note_type <= lower_note_type;
            end
            // assign same note
            else begin
               temp_vla <= prev_vla;
               last_note_type <= same_note_type;
            end
         end

         3'b101: begin
            // assign lower note
            if(rand[0]) begin
               temp_vla <= lower_note;
               last_note_type <= lower_note_type;
            end
            // assign higher note
            else begin
               temp_vla <= higher_note;
               last_note_type <= higher_note_type;
            end
         end
```

```
            3'b110: begin
               // assign same note
               if(rand[0]) begin
                  temp_vla <= prev_vla;
                  last_note_type <= same_note_type;
               end
               // assign higher note
               else begin
                  temp_vla <= higher_note;
                  last_note_type <= higher_note_type;
               end
            end

            3'b111: begin
               if(rand[0]) begin
                  // assign lower note
                  if(rand[1]) begin
                     temp_vla <= lower_note;
                     last_note_type <= lower_note_type;
                  end
                  // assign higher note
                  else begin
                     temp_vla <= higher_note;
                     last_note_type <= higher_note_type;
                  end
               end
               // assign same note
               else begin
                  temp_vla <= prev_vla;
                  last_note_type <= same_note_type;
               end
            end

            default: begin
               // assign same note
               temp_vla <= prev_vla;
               last_note_type <= same_note_type;
            end
         endcase

         search_state <= S_VLN2_UP_INITIALIZE;
      end
   end

   // based on the chord, set the note counter to the lowest
   // possible note for the violin 2
   S_VLN2_UP_INITIALIZE: begin
      // initialize the note counter
      // major chord
      if(chord[4]) case(chord[3:0])
         4'd0: begin
            note_counter <= 6'd24;
            counter_state <= S_ROOT;
         end
         4'd1: begin
            note_counter <= 6'd25;
            counter_state <= S_ROOT;
```

```
    end
    4'd2: begin
       note_counter <= 6'd26;
       counter_state <= S_ROOT;
    end
    4'd3: begin
       note_counter <= 6'd22;
       counter_state <= S_FIFTH;
    end
    4'd4: begin
       note_counter <= 6'd23;
       counter_state <= S_FIFTH;
    end
    4'd5: begin
       note_counter <= 6'd24;
       counter_state <= S_FIFTH;
    end
    4'd6: begin
       note_counter <= 6'd22;
       counter_state <= S_THIRD;
    end
    4'd7: begin
       note_counter <= 6'd23;
       counter_state <= S_THIRD;
    end
    4'd8: begin
       note_counter <= 6'd24;
       counter_state <= S_THIRD;
    end
    4'd9: begin
       note_counter <= 6'd25;
       counter_state <= S_THIRD;
    end
    4'd10: begin
       note_counter <= 6'd22;
       counter_state <= S_ROOT;
    end
    4'd11: begin
       note_counter <= 6'd23;
       counter_state <= S_ROOT;
    end
    default: begin
       note_counter <= 6'd24;
       counter_state <= S_ROOT;
    end
endcase
// minor chord
else case(chord[3:0])
    4'd0: begin
       note_counter <= 6'd24;
       counter_state <= S_ROOT;
    end
    4'd1: begin
       note_counter <= 6'd25;
       counter_state <= S_ROOT;
    end
    4'd2: begin
```

```
            note_counter <= 6'd26;
            counter_state <= S_ROOT;
         end
         4'd3: begin
            note_counter <= 6'd22;
            counter_state <= S_FIFTH;
         end
         4'd4: begin
            note_counter <= 6'd23;
            counter_state <= S_FIFTH;
         end
         4'd5: begin
            note_counter <= 6'd24;
            counter_state <= S_FIFTH;
         end
         4'd6: begin
            note_counter <= 6'd25;
            counter_state <= S_FIFTH;
         end
         4'd7: begin
            note_counter <= 6'd22;
            counter_state <= S_THIRD;
         end
         4'd8: begin
            note_counter <= 6'd23;
            counter_state <= S_THIRD;
         end
         4'd9: begin
            note_counter <= 6'd24;
            counter_state <= S_THIRD;
         end
         4'd10: begin
            note_counter <= 6'd22;
            counter_state <= S_ROOT;
         end
         4'd11: begin
            note_counter <= 6'd23;
            counter_state <= S_ROOT;
         end
         default: begin
            note_counter <= 6'd24;
            counter_state <= S_ROOT;
         end
      endcase

      // update availability and search parameters
      note_type_available[last_note_type] <=
         note_type_available[last_note_type] - 1;
      note_found <= 3'b000;
      search_state <= S_VLN2_SEARCH_UP;
   end

   // start incrementing the note counter, checking if the
   // note fits the following criteria:
   // 1. it has to be less than or equal to the previous violin 2
   //    note
   // 2. it has to fit into the chord based on the current note
```

```
//    availability
S_VLN2_SEARCH_UP: begin
   // finding nearest lower neighbor
   if(note_counter < prev_vln2) begin
      if(note_type_available[counter_state] > 0) begin
         lower_note <= note_counter;
         note_found[0] <= 1;
         lower_note_type <= counter_state;
      end
   end
   // previous note is still valid
   else if(note_counter == prev_vln2) begin
      if(note_type_available[counter_state] > 0) begin
         note_found[1] <= 1;
         same_note_type <= counter_state;
      end
   end
   // counter went above previous note
   else search_state <= S_VLN2_DOWN_INITIALIZE;
end

// the bottom-up search is done, so now set the note counter
// to the highest possible note for the violin 2
S_VLN2_DOWN_INITIALIZE: begin
   // initialize the note counter
   // major chord
   if(chord[4]) case(chord[3:0])
      4'd0: begin
         note_counter <= 6'd43;
         counter_state <= S_FIFTH;
      end
      4'd1: begin
         note_counter <= 6'd44;
         counter_state <= S_FIFTH;
      end
      4'd2: begin
         note_counter <= 6'd42;
         counter_state <= S_THIRD;
      end
      4'd3: begin
         note_counter <= 6'd43;
         counter_state <= S_THIRD;
      end
      4'd4: begin
         note_counter <= 6'd44;
         counter_state <= S_THIRD;
      end
      4'd5: begin
         note_counter <= 6'd41;
         counter_state <= S_ROOT;
      end
      4'd6: begin
         note_counter <= 6'd42;
         counter_state <= S_ROOT;
      end
      4'd7: begin
         note_counter <= 6'd43;
```

```
            counter_state <= S_ROOT;
        end
        4'd8: begin
            note_counter <= 6'd44;
            counter_state <= S_ROOT;
        end
        4'd9: begin
            note_counter <= 6'd40;
            counter_state <= S_FIFTH;
        end
        4'd10: begin
            note_counter <= 6'd41;
            counter_state <= S_FIFTH;
        end
        4'd11: begin
            note_counter <= 6'd42;
            counter_state <= S_FIFTH;
        end
        default: begin
            note_counter <= 6'd43;
            counter_state <= S_FIFTH;
        end
    endcase
    // minor chord
    else case(chord[3:0])
        4'd0: begin
            note_counter <= 6'd43;
            counter_state <= S_FIFTH;
        end
        4'd1: begin
            note_counter <= 6'd44;
            counter_state <= S_FIFTH;
        end
        4'd2: begin
            note_counter <= 6'd41;
            counter_state <= S_THIRD;
        end
        4'd3: begin
            note_counter <= 6'd42;
            counter_state <= S_THIRD;
        end
        4'd4: begin
            note_counter <= 6'd43;
            counter_state <= S_THIRD;
        end
        4'd5: begin
            note_counter <= 6'd44;
            counter_state <= S_THIRD;
        end
        4'd6: begin
            note_counter <= 6'd42;
            counter_state <= S_ROOT;
        end
        4'd7: begin
            note_counter <= 6'd43;
            counter_state <= S_ROOT;
        end
```

```
      4'd8: begin
         note_counter <= 6'd44;
         counter_state <= S_ROOT;
      end
      4'd9: begin
         note_counter <= 6'd40;
         counter_state <= S_FIFTH;
      end
      4'd10: begin
         note_counter <= 6'd41;
         counter_state <= S_FIFTH;
      end
      4'd11: begin
         note_counter <= 6'd42;
         counter_state <= S_FIFTH;
      end
      default: begin
         note_counter <= 6'd43;
         counter_state <= S_FIFTH;
      end
   endcase

   search_state <= S_VLN2_SEARCH_DOWN;
end

// start decrementing the note counter, checking if the note
// fits the following criteria:
// 1. it has to be greater than the previous violin 2 note
// 2. it has to fit into the chord based on the current note
//    availability
// once all notes greater than the previous note have been
// considered, randomly choose the next note,
// using candidates found in the bottom-up and top-down searches
S_VLN2_SEARCH_DOWN: begin
   // finding nearest upper neighbor
   if(note_counter > prev_vln2) begin
      if(note_type_available[counter_state] > 0) begin
         higher_note <= note_counter;
         note_found[2] <= 1;
         higher_note_type <= counter_state;
      end
   end
   // counter went below previous note
   else begin
      case(note_found)
         3'b001: begin
            // assign lower note
            temp_vln2 <= lower_note;
            last_note_type <= lower_note_type;
         end

         3'b010: begin
            // assign same note
            temp_vln2 <= prev_vln2;
            last_note_type <= same_note_type;
         end
```

```
3'b100: begin
   // assign higher note
   temp_vln2 <= higher_note;
   last_note_type <= higher_note_type;
end

3'b011: begin
   // assign lower note
   if(rand[0]) begin
      temp_vln2 <= lower_note;
      last_note_type <= lower_note_type;
   end
   // assign same note
   else begin
      temp_vln2 <= prev_vln2;
      last_note_type <= same_note_type;
   end
end

3'b101: begin
   // assign lower note
   if(rand[0]) begin
      temp_vln2 <= lower_note;
      last_note_type <= lower_note_type;
   end
   // assign higher note
   else begin
      temp_vln2 <= higher_note;
      last_note_type <= higher_note_type;
   end
end

3'b110: begin
   // assign same note
   if(rand[0]) begin
      temp_vln2 <= prev_vln2;
      last_note_type <= same_note_type;
   end
   // assign higher note
   else begin
      temp_vln2 <= higher_note;
      last_note_type <= higher_note_type;
   end
end

3'b111: begin
   if(rand[0]) begin
      // assign lower note
      if(rand[1]) begin
         temp_vln2 <= lower_note;
         last_note_type <= lower_note_type;
      end
      // assign higher note
      else begin
         temp_vln2 <= higher_note;
         last_note_type <= higher_note_type;
      end
```

```
            end
            // assign same note
            else begin
               temp_vln2 <= prev_vln2;
               last_note_type <= same_note_type;
            end
         end

         default: begin
            // assign same note
            temp_vln2 <= prev_vln2;
            last_note_type <= same_note_type;
         end
      endcase

      search_state <= S_VLN1_UP_INITIALIZE;
   end
end

// based on the chord, set the note counter to the lowest
// possible note on the violin
S_VLN1_UP_INITIALIZE: begin
   // initialize the note counter
   // major chord
   if(chord[4]) case(chord[3:0])
      4'd0: begin
         note_counter <= 6'd24;
         counter_state <= S_ROOT;
      end
      4'd1: begin
         note_counter <= 6'd25;
         counter_state <= S_ROOT;
      end
      4'd2: begin
         note_counter <= 6'd26;
         counter_state <= S_ROOT;
      end
      4'd3: begin
         note_counter <= 6'd22;
         counter_state <= S_FIFTH;
      end
      4'd4: begin
         note_counter <= 6'd23;
         counter_state <= S_FIFTH;
      end
      4'd5: begin
         note_counter <= 6'd24;
         counter_state <= S_FIFTH;
      end
      4'd6: begin
         note_counter <= 6'd22;
         counter_state <= S_THIRD;
      end
      4'd7: begin
         note_counter <= 6'd23;
         counter_state <= S_THIRD;
      end
```

```
   4'd8: begin
      note_counter <= 6'd24;
      counter_state <= S_THIRD;
   end
   4'd9: begin
      note_counter <= 6'd25;
      counter_state <= S_THIRD;
   end
   4'd10: begin
      note_counter <= 6'd22;
      counter_state <= S_ROOT;
   end
   4'd11: begin
      note_counter <= 6'd23;
      counter_state <= S_ROOT;
   end
   default: begin
      note_counter <= 6'd24;
      counter_state <= S_ROOT;
   end
endcase
// minor chord
else case(chord[3:0])
   4'd0: begin
      note_counter <= 6'd24;
      counter_state <= S_ROOT;
   end
   4'd1: begin
      note_counter <= 6'd25;
      counter_state <= S_ROOT;
   end
   4'd2: begin
      note_counter <= 6'd26;
      counter_state <= S_ROOT;
   end
   4'd3: begin
      note_counter <= 6'd22;
      counter_state <= S_FIFTH;
   end
   4'd4: begin
      note_counter <= 6'd23;
      counter_state <= S_FIFTH;
   end
   4'd5: begin
      note_counter <= 6'd24;
      counter_state <= S_FIFTH;
   end
   4'd6: begin
      note_counter <= 6'd25;
      counter_state <= S_FIFTH;
   end
   4'd7: begin
      note_counter <= 6'd22;
      counter_state <= S_THIRD;
   end
   4'd8: begin
      note_counter <= 6'd23;
```

```
            counter_state <= S_THIRD;
         end
         4'd9: begin
            note_counter <= 6'd24;
            counter_state <= S_THIRD;
         end
         4'd10: begin
            note_counter <= 6'd22;
            counter_state <= S_ROOT;
         end
         4'd11: begin
            note_counter <= 6'd23;
            counter_state <= S_ROOT;
         end
         default: begin
            note_counter <= 6'd24;
            counter_state <= S_ROOT;
         end
      endcase

      // update availability and search parameters
      note_type_available[last_note_type] <=
         note_type_available[last_note_type] - 1;
      note_found <= 3'b000;
      search_state <= S_VLN1_SEARCH_UP;
   end

   // start incrementing the note counter, checking if the
   // note fits the following criteria:
   // 1. it has to be less than or equal to the previous violin
   //    1 note
   // 2. it has to fit into the chord based on the current note
   //    availability
   S_VLN1_SEARCH_UP: begin
      // finding nearest lower neighbor
      if(note_counter < prev_vln1) begin
         if(note_type_available[counter_state] > 0) begin
            lower_note <= note_counter;
            note_found[0] <= 1;
            lower_note_type <= counter_state;
         end
      end
      // previous note is still valid
      else if(note_counter == prev_vln1) begin
         if(note_type_available[counter_state] > 0) begin
            note_found[1] <= 1;
            same_note_type <= counter_state;
         end
      end
      // counter went above previous note
      else search_state <= S_VLN1_DOWN_INITIALIZE;
   end

   // the bottom-up search is done, so now set the note counter
   // to the highest possible note for the violin 1
   S_VLN1_DOWN_INITIALIZE: begin
      // initialize the note counter
```

```
// major chord
if(chord[4]) case(chord[3:0])
   4'd0: begin
      note_counter <= 6'd48;
      counter_state <= S_ROOT;
   end
   4'd1: begin
      note_counter <= 6'd49;
      counter_state <= S_ROOT;
   end
   4'd2: begin
      note_counter <= 6'd50;
      counter_state <= S_ROOT;
   end
   4'd3: begin
      note_counter <= 6'd51;
      counter_state <= S_ROOT;
   end
   4'd4: begin
      note_counter <= 6'd47;
      counter_state <= S_FIFTH;
   end
   4'd5: begin
      note_counter <= 6'd48;
      counter_state <= S_FIFTH;
   end
   4'd6: begin
      note_counter <= 6'd49;
      counter_state <= S_FIFTH;
   end
   4'd7: begin
      note_counter <= 6'd50;
      counter_state <= S_FIFTH;
   end
   4'd8: begin
      note_counter <= 6'd51;
      counter_state <= S_FIFTH;
   end
   4'd9: begin
      note_counter <= 6'd49;
      counter_state <= S_THIRD;
   end
   4'd10: begin
      note_counter <= 6'd50;
      counter_state <= S_THIRD;
   end
   4'd11: begin
      note_counter <= 6'd51;
      counter_state <= S_THIRD;
   end
   default: begin
      note_counter <= 6'd48;
      counter_state <= S_ROOT;
   end
endcase
// minor chord
else case(chord[3:0])
```

```
      4'd0: begin
         note_counter <= 6'd51;
         counter_state <= S_THIRD;
      end
      4'd1: begin
         note_counter <= 6'd49;
         counter_state <= S_ROOT;
      end
      4'd2: begin
         note_counter <= 6'd50;
         counter_state <= S_ROOT;
      end
      4'd3: begin
         note_counter <= 6'd51;
         counter_state <= S_ROOT;
      end
      4'd4: begin
         note_counter <= 6'd47;
         counter_state <= S_FIFTH;
      end
      4'd5: begin
         note_counter <= 6'd48;
         counter_state <= S_FIFTH;
      end
      4'd6: begin
         note_counter <= 6'd49;
         counter_state <= S_FIFTH;
      end
      4'd7: begin
         note_counter <= 6'd50;
         counter_state <= S_FIFTH;
      end
      4'd8: begin
         note_counter <= 6'd51;
         counter_state <= S_FIFTH;
      end
      4'd9: begin
         note_counter <= 6'd48;
         counter_state <= S_THIRD;
      end
      4'd10: begin
         note_counter <= 6'd49;
         counter_state <= S_THIRD;
      end
      4'd11: begin
         note_counter <= 6'd50;
         counter_state <= S_THIRD;
      end
      default: begin
         note_counter <= 6'd51;
         counter_state <= S_THIRD;
      end
   endcase

   search_state <= S_VLN1_SEARCH_DOWN;
end
```

```
// start decrementing the note counter, checking if the note
// fits the following criteria:
// 1. it has to be greater than the previous violin 1 note
// 2. it has to fit into the chord based on the current note
//    availability
// once all notes greater than the previous note have been
// considered, randomly choose the next note,
// using candidates found in the bottom-up and top-down searches
S_VLN1_SEARCH_DOWN: begin
   // finding nearest upper neighbor
   if(note_counter > prev_vln1) begin
      if(note_type_available[counter_state] > 0) begin
         higher_note <= note_counter;
         note_found[2] <= 1;
         higher_note_type <= counter_state;
      end
   end
   // counter went below previous note
   else begin
      case(note_found)
         3'b001: begin
            // assign lower note
            temp_vln1 <= lower_note;
            last_note_type <= lower_note_type;
         end

         3'b010: begin
            // assign same note
            temp_vln1 <= prev_vln1;
            last_note_type <= same_note_type;
         end

         3'b100: begin
            // assign higher note
            temp_vln1 <= higher_note;
            last_note_type <= higher_note_type;
         end

         3'b011: begin
            // assign lower note
            if(rand[0]) begin
               temp_vln1 <= lower_note;
               last_note_type <= lower_note_type;
            end
            // assign same note
            else begin
               temp_vln1 <= prev_vln1;
               last_note_type <= same_note_type;
            end
         end

         3'b101: begin
            // assign lower note
            if(rand[0]) begin
               temp_vln1 <= lower_note;
               last_note_type <= lower_note_type;
            end
```

```
            // assign higher note
            else begin
               temp_vln1 <= higher_note;
               last_note_type <= higher_note_type;
            end
         end

         3'b110: begin
            // assign same note
            if(rand[0]) begin
               temp_vln1 <= prev_vln1;
               last_note_type <= same_note_type;
            end
            // assign higher note
            else begin
               temp_vln1 <= higher_note;
               last_note_type <= higher_note_type;
            end
         end

         3'b111: begin
            if(rand[0]) begin
               // assign lower note
               if(rand[1]) begin
                  temp_vln1 <= lower_note;
                  last_note_type <= lower_note_type;
               end
               // assign higher note
               else begin
                  temp_vln1 <= higher_note;
                  last_note_type <= higher_note_type;
               end
            end
            // assign same note
            else begin
               temp_vln1 <= prev_vln1;
               last_note_type <= same_note_type;
            end
         end

         default: begin
            // assign same note
            temp_vln1 <= prev_vln1;
            last_note_type <= same_note_type;
         end
      endcase

      search_state <= S_FINISH;
   end
end

S_FINISH: begin
   // store notes for next search
   prev_cel <= temp_cel;
   prev_vla <= temp_vla;
   prev_vln2 <= temp_vln2;
   prev_vln1 <= temp_vln1;
```

```
      // adjust note values for respective oscillators
      temp_cel <= temp_cel - 6'd3;
      temp_vla <= temp_vla - 6'd15;
      temp_vln2 <= temp_vln2 - 6'd22;
      temp_vln1 <= temp_vln1 - 6'd22;
      // wait for next chord
      search_state <= S_WAIT_FOR_CHORD;
   end

   default: search_state <= S_WAIT_FOR_CHORD;
endcase

case(counter_state)

   // current note candidate is a root of the chord
   S_ROOT: begin
      // it's doing a bottom-up search, so increment
      if(search_state == S_CEL_SEARCH_UP ||
         search_state == S_VLA_SEARCH_UP ||
         search_state == S_VLN2_SEARCH_UP ||
         search_state == S_VLN1_SEARCH_UP) begin

         // major chord
         if(chord[4] == 1) note_counter <= note_counter + 4;
         // minor chord
         else note_counter <= note_counter + 3;

         counter_state <= S_THIRD;
      end

      // it's doing a top-down search, so decrement
      else if(search_state == S_CEL_SEARCH_DOWN ||
              search_state == S_VLA_SEARCH_DOWN ||
              search_state == S_VLN2_SEARCH_DOWN ||
              search_state == S_VLN1_SEARCH_DOWN) begin

         note_counter <= note_counter - 5;
         counter_state <= S_FIFTH;
      end
   end

   // current note candidate is a third of the chord
   S_THIRD: begin
      // it's doing a bottom-up search, so increment
      if(search_state == S_CEL_SEARCH_UP ||
         search_state == S_VLA_SEARCH_UP ||
         search_state == S_VLN2_SEARCH_UP ||
         search_state == S_VLN1_SEARCH_UP) begin

         // major chord
         if(chord[4] == 1) note_counter <= note_counter + 3;
         // minor chord
         else note_counter <= note_counter + 4;

         counter_state <= S_FIFTH;
      end
```

```
            // it's doing a top-down search, so decrement
            else if(search_state == S_CEL_SEARCH_DOWN ||
                    search_state == S_VLA_SEARCH_DOWN ||
                    search_state == S_VLN2_SEARCH_DOWN ||
                    search_state == S_VLN1_SEARCH_DOWN) begin

               // major chord
               if(chord[4] == 1) note_counter <= note_counter - 4;
               // minor chord
               else note_counter <= note_counter - 3;

               counter_state <= S_ROOT;
            end
         end

         // current note candidate is a fifth of the chord
         S_FIFTH: begin
            // it's doing a bottom-up search, so increment
            if(search_state == S_CEL_SEARCH_UP ||
               search_state == S_VLA_SEARCH_UP ||
               search_state == S_VLN2_SEARCH_UP ||
               search_state == S_VLN1_SEARCH_UP) begin

               note_counter <= note_counter + 5;
               counter_state <= S_ROOT;
            end

            // it's doing a top-down search, so decrement
            else if(search_state == S_CEL_SEARCH_DOWN ||
                    search_state == S_VLA_SEARCH_DOWN ||
                    search_state == S_VLN2_SEARCH_DOWN ||
                    search_state == S_VLN1_SEARCH_DOWN) begin

               // major chord
               if(chord[4] == 1) note_counter <= note_counter - 3;
               // minor chord
               else note_counter <= note_counter - 4;

               counter_state <= S_THIRD;
            end
         end

         default: counter_state <= S_ROOT;
      endcase
   end
end

// assign outputs
assign cel_note = temp_cel[4:0];
assign vla_note = temp_vla[4:0];
assign vln2_note = temp_vln2[4:0];
assign vln1_note = temp_vln1[4:0];

endmodule
```

10 Appendix C: Verilog – Music Synthesis and Audio Modules

10.1 Violin Module

```
// Violin Module
// Author: Harley Zhang
// High-level module that includes BRAMs, oscillator,
// and envelope generator for violin

module violin(input clock,
              input reset,
              input received_audio,
              input [4:0] vln_note,
              output vln_ready,
              output signed [15:0] vln_out);

   // Original sampling frequency is 44.1kHz,
   // and (27MHz)/(44.1kHz) = 612
   parameter SAMPLE_CYCLE_COUNT = 612;

   wire [7:0] vln_addr;
   wire vln_addr_ready;
   reg vln_osc_ready;
   wire signed [15:0] vln_G_data_out;
   wire signed [15:0] vln_D_data_out;
   wire signed [15:0] vln_A_data_out;
   wire signed [15:0] vln_E_data_out;
   reg signed [15:0] to_env_gen_vln_data;
   reg vln_count_reset = 0;
   wire [31:0] vln_count;
   reg vln_sample_ready;
   reg [4:0] prev_vln_note;
   reg new_vln_note;

   // Oscillator contains instrument samples with given BRAM depths
   oscillator #(.bram_1_depth(225),.bram_2_depth(150),
                .bram_3_depth(101),.bram_4_depth(67))
              vln_osc(.clock(clock),.ready(vln_osc_ready),.note(vln_note),
                      .addr(vln_addr),.addr_ready(vln_addr_ready));
   violin_g_bram vln_G(.addr(vln_addr[7:0]),.clk(clock),
                       .dout(vln_G_data_out));
   violin_d_bram vln_D(.addr(vln_addr[7:0]),.clk(clock),
                       .dout(vln_D_data_out));
   violin_a_bram vln_A(.addr(vln_addr[6:0]),.clk(clock),
                       .dout(vln_A_data_out));
   violin_e_bram vln_E(.addr(vln_addr[6:0]),.clk(clock),
                       .dout(vln_E_data_out));

   counter_32 vln_counter(.clock(clock),.reset(vln_count_reset),
                          .count(vln_count));

   always @(posedge clock) begin

      if (vln_count == SAMPLE_CYCLE_COUNT) begin
         vln_count_reset <= 1;
```

```
            vln_osc_ready <= 1;
            vln_sample_ready <= 0;
         end

         else if (vln_addr_ready) begin

         // Get data from appropriate BRAM
            if (vln_note < 5'b00111)
               to_env_gen_vln_data <= vln_G_data_out;
            else if (vln_note < 5'b01110)
               to_env_gen_vln_data <= vln_D_data_out;
            else if (vln_note < 5'b10101)
               to_env_gen_vln_data <= vln_A_data_out;
            else
               to_env_gen_vln_data <= vln_E_data_out;

            vln_sample_ready <= 1;
            prev_vln_note <= vln_note;

            if(prev_vln_note == vln_note) new_vln_note <= 0;
            else new_vln_note <= 1;

            vln_count_reset <= 0;
            vln_osc_ready <= 0;

         end

         else begin
            vln_count_reset <= 0;
            vln_osc_ready <= 0;
            vln_sample_ready <= 0;
         end

      end

      wire signed [15:0] vln_out_temp;
      wire vln_ready_temp;

      envelope_generator vln_env_gen(.clock(clock),.reset(reset),
                                     .sample_ready(vln_sample_ready),
                                     .new_note(new_vln_note),
                                     .received_audio(received_audio),
                                     .sample(to_env_gen_vln_data),
                                     .out(vln_out_temp),
                                     .envelope_ready(vln_ready_temp));

      assign vln_out = vln_out_temp;
      assign vln_ready = vln_ready_temp;

endmodule
```

10.2 Viola Module

```
// Viola Module
// Author: Harley Zhang
// High-level module that includes BRAMs, oscillator,
// and envelope generator for viola

module viola(input clock,
             input reset,
             input received_audio,
             input [4:0] vla_note,
             output vla_ready,
             output signed [15:0] vla_out);

   // Original sampling frequency is 44.1kHz,
   // and (27MHz)/(44.1kHz) = 612
   parameter SAMPLE_CYCLE_COUNT = 612;

   wire [8:0] vla_addr;
   wire vla_addr_ready;
   reg vla_osc_ready;
   wire signed [15:0] vla_C_data_out;
   wire signed [15:0] vla_G_data_out;
   wire signed [15:0] vla_D_data_out;
   wire signed [15:0] vla_A_data_out;
   reg signed [15:0] to_env_gen_vla_data;
   reg vla_count_reset = 0;
   wire [31:0] vla_count;
   reg vla_sample_ready;
   reg [4:0] prev_vla_note;
   reg new_vla_note;

   // Oscillator contains instrument samples with given BRAM depths
   oscillator #(.bram_1_depth(338),.bram_2_depth(225),
                .bram_3_depth(150),.bram_4_depth(101))
              vla_osc(.clock(clock),.ready(vla_osc_ready),.note(vla_note),
                      .addr(vla_addr),.addr_ready(vla_addr_ready));
   viola_c_bram  vla_C(.addr(vla_addr[8:0]),.clk(clock),
                       .dout(vla_C_data_out));
   violin_g_bram vla_G(.addr(vla_addr[7:0]),.clk(clock),
                       .dout(vla_G_data_out));
   violin_d_bram vla_D(.addr(vla_addr[7:0]),.clk(clock),
                       .dout(vla_D_data_out));
   violin_a_bram vla_A(.addr(vla_addr[6:0]),.clk(clock),
                       .dout(vla_A_data_out));

   counter_32 vla_counter(.clock(clock),.reset(vla_count_reset),
                          .count(vla_count));

   always @(posedge clock) begin

      if (vla_count == SAMPLE_CYCLE_COUNT) begin
         vla_count_reset <= 1;
         vla_osc_ready <= 1;
         vla_sample_ready <= 0;
      end
```

```
      else if (vla_addr_ready) begin

         if (vla_note < 5'b00111)
            to_env_gen_vla_data <= vla_C_data_out;
         else if (vla_note < 5'b01110)
            to_env_gen_vla_data <= vla_G_data_out;
         else if (vla_note < 5'b10101)
            to_env_gen_vla_data <= vla_D_data_out;
         else
            to_env_gen_vla_data <= vla_A_data_out;

         vla_sample_ready <= 1;
         prev_vla_note <= vla_note;

         if (prev_vla_note == vla_note) new_vla_note <= 0;
         else new_vla_note <= 1;

         vla_count_reset <= 0;
         vla_osc_ready <= 0;

      end

      else begin
         vla_count_reset <= 0;
         vla_osc_ready <= 0;
         vla_sample_ready <= 0;
      end

   end

   wire signed [15:0] vla_out_temp;
   wire vla_ready_temp;

   envelope_generator vla_env_gen(.clock(clock),.reset(reset),
                                  .sample_ready(vla_sample_ready),
                                  .new_note(new_vla_note),
                                  .received_audio(received_audio),
                                  .sample(to_env_gen_vla_data),
                                  .out(vla_out_temp),
                                  .envelope_ready(vla_ready_temp));

   assign vla_out = vla_out_temp;
   assign vla_ready = vla_ready_temp;

endmodule
```

10.3 Cello Module

```
// Cello Module
// Author: Harley Zhang
// High-level module that includes BRAMs, oscillator,
// and envelope generator for cello

module cello(input clock,
             input reset,
             input received_audio,
             input [4:0] cel_note,
             output cel_ready,
             output signed [15:0] cel_out);

   // Original sampling frequency is 44.1kHz,
   // and (27MHz)/(44.1kHz) = 612
   parameter SAMPLE_CYCLE_COUNT = 612;

   wire [9:0] cel_addr;
   wire cel_addr_ready;
   reg cel_osc_ready;
   wire signed [15:0] cel_C_data_out;
   wire signed [15:0] cel_G_data_out;
   wire signed [15:0] cel_D_data_out;
   wire signed [15:0] cel_A_data_out;
   reg signed [15:0] to_env_gen_cel_data;
   reg cel_count_reset = 0;
   wire [31:0] cel_count;
   reg cel_sample_ready;
   reg [4:0] prev_cel_note;
   reg new_cel_note;

   // Oscillator contains instrument samples with given BRAM depths
   oscillator #(.bram_1_depth(675),.bram_2_depth(448),
                .bram_3_depth(301),.bram_4_depth(201))
              cel_osc(.clock(clock),.ready(cel_osc_ready),.note(cel_note),
                      .addr(cel_addr),.addr_ready(cel_addr_ready));
   cello_c_bram cel_C(.addr(cel_addr[9:0]),.clk(clock),
                      .dout(cel_C_data_out));
   cello_g_bram cel_G(.addr(cel_addr[8:0]),.clk(clock),
                      .dout(cel_G_data_out));
   cello_d_bram cel_D(.addr(cel_addr[8:0]),.clk(clock),
                      .dout(cel_D_data_out));
   cello_a_bram cel_A(.addr(cel_addr[7:0]),.clk(clock),
                      .dout(cel_A_data_out));

   counter_32 cel_counter(.clock(clock),.reset(cel_count_reset),
                          .count(cel_count));

   always @(posedge clock) begin

      if (cel_count == SAMPLE_CYCLE_COUNT) begin
         cel_count_reset <= 1;
         cel_osc_ready <= 1;
         cel_sample_ready <= 0;
      end
```

```
      else if (cel_addr_ready) begin

         if (cel_note < 5'b00111)
            to_env_gen_cel_data <= cel_C_data_out;
         else if (cel_note < 5'b01110)
            to_env_gen_cel_data <= cel_G_data_out;
         else if (cel_note < 5'b10101)
            to_env_gen_cel_data <= cel_D_data_out;
         else
            to_env_gen_cel_data <= cel_A_data_out;

         cel_sample_ready <= 1;
         prev_cel_note <= cel_note;

         if(prev_cel_note == cel_note) new_cel_note <= 0;
         else new_cel_note <= 1;

         cel_count_reset <= 0;
         cel_osc_ready <= 0;

      end

      else begin
         cel_count_reset <= 0;
         cel_osc_ready <= 0;
         cel_sample_ready <= 0;
      end

   end

   wire signed [15:0] cel_out_temp;
   wire cel_ready_temp;

   envelope_generator cel_env_gen(.clock(clock),.reset(reset),
                                  .sample_ready(cel_sample_ready),
                                  .new_note(new_cel_note),
                                  .received_audio(received_audio),
                                  .sample(to_env_gen_cel_data),
                                  .out(cel_out_temp),
                                  .envelope_ready(cel_ready_temp));

   assign cel_out = cel_out_temp;
   assign cel_ready = cel_ready_temp;


endmodule
```

10.4 Oscillator

```
// Oscillator module
// Author: Harley Zhang
// Takes note and accesses BRAM with corresponding frequency
// Interval = note freq / stored note freq

module oscillator #(parameter bram_1_depth = 256,
                    parameter bram_2_depth = 256,
                                parameter bram_3_depth = 256,
                                parameter bram_4_depth = 256)
                   (input clock, ready,
                    input [4:0] note,
                    output [9:0] addr,
                    output reg addr_ready);

   reg [19:0] prev_1_addr = 0;
   reg [19:0] prev_2_addr = 0;
   reg [19:0] prev_3_addr = 0;
   reg [19:0] prev_4_addr = 0;
   reg [19:0] interval;

   assign addr = ((note < 5'b00111) ? prev_1_addr[19:10] :
                 ((note < 5'b01110) ? prev_2_addr[19:10] :
                 ((note < 5'b10101) ? prev_3_addr[19:10] :
                                      prev_4_addr[19:10])));

   always @(posedge clock) begin

      if (ready) begin

         case (note)

            5'b00000: interval <= 20'b00000_00001_00000_00000;
            5'b00001: interval <= 20'b00000_00001_00001_11101;
            5'b00010: interval <= 20'b00000_00001_00011_11101;
            5'b00011: interval <= 20'b00000_00001_00110_00010;
            5'b00100: interval <= 20'b00000_00001_01000_01010;
            5'b00101: interval <= 20'b00000_00001_01010_10111;
            5'b00110: interval <= 20'b00000_00001_01101_01000;

            5'b00111: interval <= 20'b00000_00001_00000_00000;
            5'b01000: interval <= 20'b00000_00001_00001_11101;
            5'b01001: interval <= 20'b00000_00001_00011_11101;
            5'b01010: interval <= 20'b00000_00001_00110_00010;
            5'b01011: interval <= 20'b00000_00001_01000_01010;
            5'b01100: interval <= 20'b00000_00001_01010_10111;
            5'b01101: interval <= 20'b00000_00001_01101_01000;

            5'b01110: interval <= 20'b00000_00001_00000_00000;
            5'b01111: interval <= 20'b00000_00001_00001_11101;
            5'b10000: interval <= 20'b00000_00001_00011_11101;
            5'b10001: interval <= 20'b00000_00001_00110_00010;
            5'b10010: interval <= 20'b00000_00001_01000_01010;
            5'b10011: interval <= 20'b00000_00001_01010_10111;
            5'b10100: interval <= 20'b00000_00001_01101_01000;
```

```
      5'b10101: interval <= 20'b00000_00001_00000_00000;
      5'b10110: interval <= 20'b00000_00001_00001_11101;
      5'b10111: interval <= 20'b00000_00001_00011_11101;
      5'b11000: interval <= 20'b00000_00001_00110_00010;
      5'b11001: interval <= 20'b00000_00001_01000_01010;
      5'b11010: interval <= 20'b00000_00001_01010_10111;
      5'b11011: interval <= 20'b00000_00001_01101_01000;
      5'b11100: interval <= 20'b00000_00001_01111_11110;
      5'b11101: interval <= 20'b00000_00001_10010_11001;
      5'b11110: interval <= 20'b00000_00001_10101_11010;
      5'b11111: interval <= 20'b00000_00001_11001_00001;

      default:  interval <= 0;

   endcase

   if (note < 5'b00111) begin
      if (prev_1_addr + interval >= (bram_1_depth << 10)) begin
         prev_1_addr <= prev_1_addr + interval - (bram_1_depth << 10);
      end
      else begin
         prev_1_addr <= prev_1_addr + interval;
      end
   end

   else if (note < 5'b01110) begin
      if (prev_2_addr + interval >= (bram_2_depth << 10)) begin
         prev_2_addr <= prev_2_addr + interval - (bram_2_depth << 10);
      end
      else begin
         prev_2_addr <= prev_2_addr + interval;
      end
   end

   else if (note < 5'b10101) begin
      if (prev_3_addr + interval >= (bram_3_depth << 10)) begin
         prev_3_addr <= prev_3_addr + interval - (bram_3_depth << 10);
      end
      else begin
         prev_3_addr <= prev_3_addr + interval;
      end
   end

   else begin
      if (prev_4_addr + interval >= (bram_4_depth << 10)) begin
         prev_4_addr <= prev_4_addr + interval - (bram_4_depth << 10);
      end
      else begin
         prev_4_addr <= prev_4_addr + interval;
      end
   end

   addr_ready <= 1;

end
```

```
      else addr_ready <= 0;
   end
endmodule
```

10.5 Envelope Generator

```
// envelope_generator.v
// Author: Rajeev Nayak
// The envelope_generator module scales the input sample to
// mimic the amplitude envelope of a bowed string instrument
// with vibrato.

module envelope_generator(input clock,
                          input reset,
                          input sample_ready,
                          input new_note,
                          input received_audio,
                          input signed [15:0] sample,
                          output reg signed [15:0] out,
                          output reg envelope_ready);

   // envelope generator state
   parameter S_WAIT_FOR_SAMPLE = 0;
   parameter S_ADJUST_ENVELOPE = 1;
   parameter S_APPLY_ENVELOPE = 2;
   parameter S_WAIT_FOR_MIXER = 3;

   // adsr envelope state
   parameter S_ATTACK = 0;
   parameter S_DECAY = 1;
   parameter S_SUSTAIN = 2;

   // adsr envelope parameters
   parameter ATTACK_DURATION = 18'd32768;
   parameter DECAY_DURATION = 18'd16384;
   parameter VIBRATO_DURATION = 18'd4096;
   parameter AMPLITUDE_SHIFT = 5'd15;

   reg [1:0] generator_state;
   reg [1:0] envelope_state;
   reg [17:0] envelope_timer;
   reg [17:0] vibrato_timer;
   reg vibrato_direction;
   reg signed [33:0] temp_sample;
   reg signed [33:0] temp_out;

   always @(posedge clock) begin
      if (reset) begin
         out <= 0;
         envelope_ready <= 0;
         generator_state <= S_WAIT_FOR_SAMPLE;
         envelope_state <= S_ATTACK;
         envelope_timer <= 0;
         vibrato_timer <= 0;
         vibrato_direction <= 1;
      end
      else case(generator_state)

         // wait for the next instrument sample from the BRAM
         S_WAIT_FOR_SAMPLE: begin
```

```
   envelope_ready <= 0;
   if(sample_ready) begin
      // store received sample
      temp_sample <= sample;
      // if it's a new note, go to the attack state and
      // set the amplitude appropriately
      if(new_note) begin
         envelope_state <= S_ATTACK;
         case(envelope_state)
            S_ATTACK: envelope_timer <= envelope_timer;
            S_DECAY: envelope_timer <= ATTACK_DURATION -
                                       envelope_timer;
            S_SUSTAIN: envelope_timer <= ATTACK_DURATION -
                                         DECAY_DURATION +
                                         (VIBRATO_DURATION >> 1);
         endcase
      end

      generator_state <= S_ADJUST_ENVELOPE;
   end
end

// see if the envelope state needs to transition based on the timer
S_ADJUST_ENVELOPE: begin
   case(envelope_state)
      S_ATTACK: begin
         if(envelope_timer == ATTACK_DURATION) begin
            envelope_state <= S_DECAY;
            envelope_timer <= 0;
         end
      end

      S_DECAY: begin
         if(envelope_timer == DECAY_DURATION) begin
            envelope_state <= S_SUSTAIN;
            envelope_timer <= 0;
            vibrato_timer <= 0;
            vibrato_direction <= 1;
         end
      end

      S_SUSTAIN: begin
         // change the vibrato direction if necessary
         if(vibrato_timer == 0)
            vibrato_direction <= 1;
         else if(vibrato_timer == VIBRATO_DURATION)
            vibrato_direction <= 0;
         envelope_state <= S_SUSTAIN;
      end

      default: envelope_state <= S_ATTACK;
   endcase

   generator_state <= S_APPLY_ENVELOPE;
end

// scale the sample based on the envelope state
```

```
         S_APPLY_ENVELOPE: begin
            case(envelope_state)
               S_ATTACK: begin
                  temp_out <=
                     (temp_sample * envelope_timer) >> AMPLITUDE_SHIFT;
                  envelope_timer <= envelope_timer + 2;
               end

               S_DECAY: begin
                  temp_out <=
                     (temp_sample * (ATTACK_DURATION - envelope_timer)) >>
                        AMPLITUDE_SHIFT;
                  envelope_timer <= envelope_timer + 2;
               end

               S_SUSTAIN: begin
                  temp_out <=
                     (temp_sample * (ATTACK_DURATION - DECAY_DURATION +
                                     vibrato_timer)) >> AMPLITUDE_SHIFT;
                  if(vibrato_direction) vibrato_timer <= vibrato_timer + 1;
                  else vibrato_timer <= vibrato_timer - 1;
               end

               default: temp_out <= 0;
            endcase

            generator_state <= S_WAIT_FOR_MIXER;
         end

         // wait for the mixer to receive all envelope outputs
         // and add them together
         S_WAIT_FOR_MIXER: begin
            // set output and enable ready signal
            out <= temp_out;
            envelope_ready <= 1;
            if(received_audio) generator_state <= S_WAIT_FOR_SAMPLE;
         end

      endcase
   end

endmodule
```

10.6 Mixer

```
// mixer.v
// Author: Rajeev Nayak
// The mixer module combines the 4 instrument outputs by
// adding them together and sends them to the AC97 output.

module mixer(input clock,
             input reset,
             input vln1_ready,
             input vln2_ready,
             input vla_ready,
             input cel_ready,
             input signed [15:0] vln1_out,
             input signed [15:0] vln2_out,
             input signed [15:0] vla_out,
             input signed [15:0] cel_out,
             output reg signed [17:0] audio_out_data,
             output reg received_audio);

   parameter S_WAIT_FOR_ENVELOPES = 0;
   parameter S_ADD_CEL = 1;
   parameter S_ADD_VLA = 2;
   parameter S_ADD_VLN2 = 3;
   parameter S_ADD_VLN1 = 4;
   parameter S_OUTPUT = 5;

   reg [17:0] temp_out_data;
   reg [2:0] state;

   always @(posedge clock) begin
      if(reset) begin
         temp_out_data <= 0;
         audio_out_data <= 0;
         received_audio <= 0;
         state <= S_WAIT_FOR_ENVELOPES;
      end
      else case(state)

         // wait for all 4 envelope generators to finish scaling
         // their samples
         S_WAIT_FOR_ENVELOPES: begin
            temp_out_data <= 0;
            received_audio <= 0;
            if(vln1_ready && vln2_ready && vla_ready && cel_ready)
               state <= S_ADD_CEL;
         end

         // add the cello sample to the temporary register
         S_ADD_CEL: begin
            temp_out_data <= temp_out_data + cel_out;
            state <= S_ADD_VLA;
         end

         // add the viola sample to the temporary register
         S_ADD_VLA: begin
```

```
            temp_out_data <= temp_out_data + vla_out;
            state <= S_ADD_VLN2;
         end

         // add the violin 2 sample to the temporary register
         S_ADD_VLN2: begin
            temp_out_data <= temp_out_data + vln2_out;
            state <= S_ADD_VLN1;
         end

         // add the violin 1 sample to the temporary register
         S_ADD_VLN1: begin
            temp_out_data <= temp_out_data + vln1_out;
            state <= S_OUTPUT;
         end

         // all 4 signals have been added, so set the AC97 output
         // data and inform the envelope generators that it's done
         S_OUTPUT: begin
            audio_out_data <= temp_out_data;
            received_audio <= 1;
            state <= S_WAIT_FOR_ENVELOPES;
         end

         default: state <= S_WAIT_FOR_ENVELOPES;

      endcase
   end

endmodule
```

10.7 AC97 Driver Modules

```
// stereo_audio.v
// Author: 6.111 Staff (minor edits by Rajeev Nayak)
// The stereo_audio module sends the 18-bit audio_out data to the AC97
output.
// The ac97 module assembles and disassembles the AC97 serial frames.
// The ac97commands module issues initialization commands to the AC97.

module stereo_audio (input wire clock_27mhz,
                     input wire reset,
                     input wire [4:0] volume,
                     input wire [17:0] audio_out_data,
                     output wire ready,
                     output reg audio_reset_b,   // ac97 interface signals
                     output wire ac97_sdata_out,
                     input wire ac97_sdata_in,
                     output wire ac97_synch,
                     input wire ac97_bit_clock);

   wire [7:0] command_address;
   wire [15:0] command_data;
   wire command_valid;
   wire [19:0] left_in_data, right_in_data;
   wire [19:0] left_out_data, right_out_data;

   // wait a little before enabling the AC97 codec
   reg [9:0] reset_count;
   always @(posedge clock_27mhz) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end
      else if (reset_count == 1023) audio_reset_b = 1'b1;
      else reset_count = reset_count + 1;
   end

   wire ac97_ready;
   ac97 ac97(.ready(ac97_ready),
             .command_address(command_address),
             .command_data(command_data),
             .command_valid(command_valid),
             .left_data(left_out_data), .left_valid(1'b1),
             .right_data(right_out_data), .right_valid(1'b1),
             .left_in_data(left_in_data), .right_in_data(right_in_data),
             .ac97_sdata_out(ac97_sdata_out),
             .ac97_sdata_in(ac97_sdata_in),
             .ac97_synch(ac97_synch),
             .ac97_bit_clock(ac97_bit_clock));

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

```
   reg [17:0] out_data;
   always @ (posedge clock_27mhz) if (ready) out_data <= audio_out_data;
   assign left_out_data = {out_data, 2'b00};
   assign right_out_data = left_out_data;

   // generate repeating sequence of read/writes to AC97 registers
   ac97commands cmds(.clock(clock_27mhz), .ready(ready),
                     .command_address(command_address),
                     .command_data(command_data),
                     .command_valid(command_valid),
                     .volume(volume),
                     .source(3'b000));     // mic
endmodule

module ac97 (output reg ready,
             input wire [7:0] command_address,
             input wire [15:0] command_data,
             input wire command_valid,
             input wire [19:0] left_data,
             input wire left_valid,
             input wire [19:0] right_data,
             input wire right_valid,
             output reg [19:0] left_in_data, right_in_data,
             output reg ac97_sdata_out,
             input wire ac97_sdata_in,
             output reg ac97_synch,
             input wire ac97_bit_clock);

   reg [7:0] bit_count;

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

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

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

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

```
always @(posedge ac97_bit_clock) begin
   // Generate the sync signal
   if (bit_count == 255) ac97_synch <= 1'b1;
   if (bit_count == 15) ac97_synch <= 1'b0;

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

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

   if ((bit_count >= 0) && (bit_count <= 15))
      // Slot 0: Tags
      case (bit_count[3:0])
         4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
         4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
         4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
         4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
         4'h4: ac97_sdata_out <= l_right_v; // Right data valid
         default: ac97_sdata_out <= 1'b0;
      endcase
   else if ((bit_count >= 16) && (bit_count <= 35))
      // Slot 1: Command address (8-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;
   else if ((bit_count >= 36) && (bit_count <= 55))
      // Slot 2: Command data (16-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;
   else if ((bit_count >= 56) && (bit_count <= 75)) begin
      // Slot 3: Left channel
      ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
      l_left_data <= { l_left_data[18:0], l_left_data[19] };
   end
   else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
      ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
   else ac97_sdata_out <= 1'b0;

   bit_count <= bit_count+1;
end // always @ (posedge ac97_bit_clock)

always @(negedge ac97_bit_clock) begin
   if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
   else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
```

```
         right_in_data <= { right_in_data[18:0], ac97_sdata_in };
   end
endmodule

module ac97commands (input wire clock,
                     input wire ready,
                     output wire [7:0] command_address,
                     output wire [15:0] command_data,
                     output reg command_valid,
                     input wire [4:0] volume,
                     input wire [2:0] source);

   reg [23:0] command;

   reg [3:0] state;
   initial begin
      command <= 4'h0;
      // synthesis attribute init of command is "0";
      command_valid <= 1'b0;
      // synthesis attribute init of command_valid is "0";
      state <= 16'h0000;
      // synthesis attribute init of state is "0000";
   end

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

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

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

      case (state)
         4'h0: // Read ID
         begin
            command <= 24'h80_0000;
            command_valid <= 1'b1;
         end
         4'h1: // Read ID
            command <= 24'h80_0000;
         4'h3: // headphone volume
            command <= { 8'h04, 3'b000, vol, 3'b000, vol };
         4'h5: // PCM volume
            command <= 24'h18_0808;
         4'h6: // Record source select
            command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
         4'h7: // Record gain = max
            command <= 24'h1C_0F0F;
         4'h9: // set +20db mic gain
            command <= 24'h0E_8048;
         4'hA: // Set beep volume
            command <= 24'h0A_0000;
         4'hB: // PCM out bypass mix1
            command <= 24'h20_8000;
         default: command <= 24'h80_0000;
      endcase // case(state)
```

```
   end // always @ (posedge clock)
endmodule // ac97commands
```

11 Appendix D: Verilog – Video Output Modules

11.1 XVGA Module

```
// Generates XVGA display signals (1024 x 768 @ 60Hz)
// Author: 6.111 staff

module xvga(input vclock,
            output reg [10:0] hcount,    // pixel number on current line
            output reg [9:0] vcount,     // line number
            output reg vsync,hsync,blank);

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

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

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;

   always @(posedge vclock) begin

      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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

   end

endmodule
```

11.2 Music Information Module

```
// Music Info Module
// Author: Harley Zhang
// Produces video output in music info mode, including tempo in
// beats per minute and the present chord

module music_info (input vclock,
                   input [10:0] hcount,
                   input [9:0] vcount,
                   input [4:0] chord,
                   input [31:0] tempo_period,
                   output [2:0] mpixel);

   wire [55:0] cstring1 = "TEMPO: ";
   wire [2:0]  cd1pixel;
   char_string_display cd1(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                           .pixel(cd1pixel),.cstring(cstring1),.cx(11'd352),
                           .cy(10'd300));

   wire [55:0] cstring2 = "CHORD: ";
   wire [2:0]  cd2pixel;
   char_string_display cd2(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                           .pixel(cd2pixel),.cstring(cstring2),.cx(11'd352),
                           .cy(10'd400));

   wire [55:0] tempostring;
   assign tempostring[31:0] = " BPM";
   wire [2:0] cd3pixel;
   wire [3:0] bpm_100, bpm_10, bpm_1;

   cycles2bpm
cyc2bpm1(.clock(vclock),.cycles(tempo_period),.bpm_100(bpm_100),
                        .bpm_10(bpm_10),.bpm_1(bpm_1));
   dec2char d2c1(.clock(vclock),.dec(bpm_100),.char(tempostring[55:48]),
                 .highbit(1));
   dec2char d2c2(.clock(vclock),.dec(bpm_10),.char(tempostring[47:40]),
                 .highbit(0));
   dec2char d2c3(.clock(vclock),.dec(bpm_1),.char(tempostring[39:32]),
                 .highbit(0));
   char_string_display cd3(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                           .pixel(cd3pixel),.cstring(tempostring),
                           .cx(11'd496),.cy(10'd300));


   wire [63:0] keystring;
   wire [2:0] cd4pixel;
   chord2chars c2c(.clock(vclock),.chord(chord),.chars(keystring));
   char_string_display cd4(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                           .pixel(cd4pixel),.cstring(keystring),.cx(11'd496),
                           .cy(10'd400));

   assign mpixel = cd1pixel | cd2pixel | cd3pixel | cd4pixel;

endmodule
```

11.3 String Display Module

```
// C String Display Module
// Author: I. Chuang, C. Terman
// Displays an ASCII encoded character string in a video window at some
// specified x,y pixel location.

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

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

   input vclock;  // 65MHz clock
   input [10:0] hcount; // horizontal index of current pixel (0..1023)
   input [9:0]    vcount; // vertical index of current pixel (0..767)
   output [2:0] pixel;  // char display's pixel
   input [NCHAR*8-1:0] cstring;     // character string to display
   input [10:0] cx;
   input [9:0]    cy;

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

   wire [10:0]    hoff = hcount-1-cx;
   wire [9:0]     voff = vcount-cy;
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // < NCHAR
   wire [2:0]     h = hoff[3:1];            // 0 .. 7
   wire [3:0]     v = voff[4:1];            // 0 .. 11

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

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

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

endmodule
```

11.4 Cycles-to-Decimal BPM Converter

```
// Cycle count to beats per minute conversion module
// Author: Harley Zhang
// Used to convert tempo_period in number of cycles to
// beats per minute (decimal)
// Performs division by 100, 10, and 1 by recursively subtracting

module cycles2bpm(input [31:0] cycles,
                  input clock,
                  output reg [3:0] bpm_100,
                  output reg [3:0] bpm_10,
                  output reg [3:0] bpm_1);

   // 1.62e9 = 60 * 27e6, the number of cycles per minute
   reg [33:0] div_temp = 34'd1620000000;

   reg [9:0] bpm_temp = 0;
   reg [3:0] bpm_100_count = 0, bpm_10_count = 0, bpm_1_count = 0;
   reg dividing = 0, calculating = 0;
   reg [31:0] prev_cycles;


   always @(posedge clock) begin

      if ((cycles != prev_cycles) && ~dividing && ~calculating) begin
         dividing <= 1;
         calculating <= 0;
      end
      else if ((div_temp < cycles) && dividing && ~calculating) begin
         div_temp <= 34'd1620000000;
         dividing <= 0;
         calculating <= 1;
      end
      else if ((div_temp >= cycles) && dividing && ~calculating) begin
      // Divide the number of cycles per minute by the input cycle count
         div_temp <= div_temp - cycles;
         bpm_temp <= bpm_temp + 1;
         dividing <= 1;
         calculating <= 0;
      end
      else if ((bpm_temp == 0) && ~dividing && calculating) begin
      // Outputs the 100s digit, 10s digit, and 1s digit
         bpm_100 <= bpm_100_count;
         bpm_10  <= bpm_10_count;
         bpm_1   <= bpm_1_count;
         bpm_100_count <= 0;
         bpm_10_count  <= 0;
         bpm_1_count   <= 0;
         dividing <= 0;
         calculating <= 0;
         prev_cycles <= cycles;
      end
      else if ((bpm_temp >= 100) && ~dividing && calculating) begin
         bpm_temp <= bpm_temp - 100;
         bpm_100_count <= bpm_100_count + 1;
```

```
            dividing <= 0;
            calculating <= 1;
        end
        else if ((bpm_temp >= 10) && ~dividing && calculating) begin
            bpm_temp <= bpm_temp - 10;
            bpm_10_count <= bpm_10_count + 1;
            dividing <= 0;
            calculating <= 1;
        end
        else if ((bpm_temp >= 1) && ~dividing && calculating) begin
            bpm_temp <= bpm_temp - 1;
            bpm_1_count <= bpm_1_count + 1;
            dividing <= 0;
            calculating <= 1;
        end
        else begin
            calculating <= 0;
            dividing <= 0;
        end


    end

endmodule
```

11.5 Decimal Digit-to-Character Converter

```
// Decimal to character conversion module
// Author: Harley Zhang
// Takes in a 4-bit decimal digit and outputs corresponding character
// 100s digit is not displayed when it is 0

module dec2char(input clock,
                input highbit,
                input [3:0] dec,
                output reg [7:0] char);

   always @(posedge clock) begin

      case (dec)
         1: char <= "1";
         2: char <= "2";
         3: char <= "3";
         4: char <= "4";
         5: char <= "5";
         6: char <= "6";
         7: char <= "7";
         8: char <= "8";
         9: char <= "9";
         0: char <= (highbit) ? " " : "0";
         default: char <= " ";
      endcase

   end

endmodule
```

11.6 Chord-to-Characters Converter

```
// Chord to characters conversion module
// Author: Harley Zhang
// Takes in a 5-bit chord and outputs corresponding characters

module chord2chars(input clock,
                   input [4:0] chord,
                   output reg [63:0] chars);

   always @(posedge clock) begin

      case (chord[4])
         0: chars[47:0] <= " MINOR";
         1: chars[47:0] <= " MAJOR";
         default: chars[47:0] <= "      ";
      endcase

      case (chord[3:0])
         0: chars[63:48] <= "A ";
         1: chars[63:48] <= "A#";
         2: chars[63:48] <= "B ";
         3: chars[63:48] <= "C ";
         4: chars[63:48] <= "C#";
         5: chars[63:48] <= "D ";
         6: chars[63:48] <= "D#";
         7: chars[63:48] <= "E ";
         8: chars[63:48] <= "F ";
         9: chars[63:48] <= "F#";
         10: chars[63:48] <= "G ";
         11: chars[63:48] <= "G#";
         default: chars[63:48] <= "  ";
      endcase

   end

endmodule
```

11.7 Visualization Module

```
// Visualization Module
// Author: Harley Zhang
// Displays pattern of footprints triggered off assertion of input

module visual (input vclock,           // 65MHz clock
               input reset,            // 1 to initialize module
               input [10:0] hcount,    // horizontal index of pixel (0..1023)
               input [9:0]  vcount,    // vertical index of pixel (0..767)
               input hsync,            // XVGA horiz. sync signal (active
low)
               input vsync,            // XVGA vert. sync signal (active low)
               input blank,            // XVGA blanking
                                       // (1 means output black pixel)
               input trigger,          // triggers image of footprint

               output phsync,          // visualization's horizontal sync
               output pvsync,          // visualization's vertical sync
               output pblank,          // visualization's blanking
               output [7:0] out_r,     // red output
               output [7:0] out_g,     // green output
               output [7:0] out_b);    // blue output

   // Image data
   parameter picture_w = 47;
   parameter picture_h = 113;
   parameter max_x = 1023;
   parameter max_y = 767;

   wire [7:0] out_r_temp_l,out_g_temp_l,out_b_temp_l,
              out_r_temp_r,out_g_temp_r,out_b_temp_r;
   reg [4:0] count;
   reg prev_vsync;
   reg prev_triggered;
   reg foot_sel = 0;

   reg trigger_rise = 0;

   // Initial footprint image locations (upper left pixels)
   reg [10:0] x_l = 11'd400;
   reg [9:0] y_l = 10'd300;
   reg [10:0] x_r = 11'd623;
   reg [9:0] y_r = 10'd187;

   assign phsync = hsync;
   assign pvsync = vsync;
   assign pblank = blank;
   assign out_r = foot_sel ? out_r_temp_r : out_r_temp_l;
   assign out_g = foot_sel ? out_g_temp_r : out_g_temp_l;
   assign out_b = foot_sel ? out_b_temp_r : out_b_temp_l;

   footprint_l foot_l(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                      .count(count),.x(x_l),.y(y_l),.r(out_r_temp_l),
                      .g(out_g_temp_l),.b(out_b_temp_l));
```

```
    footprint_r foot_r(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                       .count(count),.x(x_r),.y(y_r),.r(out_r_temp_r),
                       .g(out_g_temp_r),.b(out_b_temp_r));

    always @(posedge vclock) begin

       prev_vsync <= vsync;
       prev_triggered <= trigger;

       if (reset) begin
       // Reset to initial default positions
          count <= 0;
          foot_sel <= 0;
          x_l <= 11'd400;
          y_l <= 10'd300;
          x_r <= 11'd623;
          y_r <= 10'd187;
          trigger_rise <= 0;
       end

       else if (trigger && ~prev_triggered) begin
          trigger_rise <= 1;
       end

       else if (~vsync && prev_vsync) begin

          trigger_rise <= 0;

          if (trigger_rise) begin
          // Update picture locations and switch feet
             count <= 0;
             if (foot_sel == 0) begin
                if (y_l >= (picture_h << 1)) y_l <= y_l - (picture_h << 1);
                else y_l <= max_y - (picture_h << 1);
             end
             else begin
                if (y_r >= (picture_h << 1)) y_r <= y_r - (picture_h << 1);
                else y_r <= max_y - picture_h;
             end
             foot_sel <= ~foot_sel;
          end

          else if (count == 5'b11111) count <= count;

          else count <= count + 1;

       end

    end

endmodule
```

11.8 Left Footprint Module

```
// Left footprint module
// Author: Harley Zhang
// Generates image of left footprint for visualization module

module footprint_l (input vclock,
                    input reset,
                    input [4:0] count,
                    input [10:0] hcount,
                    input [10:0] x,
                    input [9:0] vcount,
                    input [9:0] y,
                    output reg [7:0] r,g,b);

   parameter picture_w = 47;
   parameter picture_h = 113;
   parameter picture_pixels = 5311;

   reg [12:0] addr = 0;
   wire bram_bit;

   foot_l_bram foot(.addr(addr),.clk(vclock),.dout(bram_bit));

   always @(posedge vclock) begin

      if (reset) begin
         addr <= 0;
      end

      else if ((hcount < x) || (hcount >= x + picture_w) ||
               (vcount < y) || (vcount >= y + picture_h)) begin
         r <= 0;
         g <= 0;
         b <= 0;
      end

      else begin

         if (bram_bit) begin
         // Image fades as count increases
            r <= 248 - (count << 3);
            b <= 248 - (count << 3);
            g <= 248 - (count << 3);
         end
         else begin
            r <= 0;
            g <= 0;
            b <= 0;
         end
         if (addr == picture_pixels - 1) addr <= 0;
         else addr <= addr + 1;

      end
   end
endmodule
```

11.9 Right Footprint Module

```
// Right footprint module
// Author: Harley Zhang
// Generates image of right footprint for visualization module

module footprint_r (input vclock,
                    input reset,
                    input [4:0] count,
                    input [10:0] hcount,
                    input [10:0] x,
                    input [9:0] vcount,
                    input [9:0] y,
                    output reg [7:0] r,g,b);

   parameter picture_w = 47;
   parameter picture_h = 113;
   parameter picture_pixels = 5311;

   reg [12:0] addr = 0;
   wire bram_bit;

   foot_r_bram foot(.addr(addr),.clk(vclock),.dout(bram_bit));

   always @(posedge vclock) begin

      if (reset) begin
         addr <= 0;
      end

      if ((hcount < x) || (hcount >= x + picture_w) ||
          (vcount < y) || (vcount >= y + picture_h)) begin
         r <= 0;
         g <= 0;
         b <= 0;
      end

      else begin

         if (bram_bit) begin
         // Image fades as count increases
            r <= 248 - (count << 3);
            b <= 248 - (count << 3);
            g <= 248 - (count << 3);
         end
         else begin
            r <= 0;
            g <= 0;
            b <= 0;
         end

         if (addr == picture_pixels - 1) addr <= 0;
         else addr <= addr + 1;
      end
   end
endmodule
```

12 Appendix E: Verilog – Top Level and Miscellaneous Modules

12.1 Debouncer

```
// Switch Debounce Module
// Author: 6.111 staff
// Produces a synchronous, debounced output

module debounce #(parameter DELAY=270000)   // .01 sec with a 27Mhz clock
                 (input reset,
                  input clock,
                  input noisy,
                  output reg clean);

   reg [23:0] count;
   reg new;

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

endmodule
```

12.2 Synchronizer

```
// Pulse synchronizer
// Author: 6.111 staff, modified by Harley Zhang

module synchronize #(parameter WIDTH = 1)  // width of data
                    (input clock,
                     input [WIDTH-1:0] in,
                     output reg [WIDTH-1:0] out);

   reg [WIDTH-1:0] sync;

   always @ (posedge clock) begin

      out <= sync;
      sync <= in;

   end
endmodule
```

12.3 32-Bit Counter

```
// 32-bit counter
// Author: 6.111 staff

module counter_32 (clock, reset, count);

   input clock, reset;
   output [31:0] count;

   reg [31:0] count = 0;

   always @(posedge clock) begin
      if (reset) count <= 0;
      else count <= count+1;
   end

endmodule
```

12.4 Top-Level Module

```
`default_nettype none

//////////////////////////////////////////////////////////////////////////////
//
//
// Musical Feet: A Step-by-Step Approach to Music Generation
//
// Top Level Module
//
// Authors: Rajeev Nayak and Harley Zhang
//
// 6.111 Fall 2008
//
//////////////////////////////////////////////////////////////////////////////
//


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

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

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

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

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

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

               clock_feedback_out, clock_feedback_in,

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

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

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

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

          switch,

          led,

          user1, user2, user3, user4,

          daughtercard,

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

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

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
```

```
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

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

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

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

   inout [43:0] daughtercard;

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

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


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

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

   // Audio Input and Output

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

   // 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
   assign led[7:0] = 8'b11111111;
   // 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 user1[31:1] = 31'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;

   // 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(.reset(power_on_reset),.clock(clock_65mhz),
                .noisy(~button_enter),.clean(user_reset));
   assign reset = user_reset | power_on_reset;
```

```
//////////////////////////////////////////////////////////////////////
   //
   // Input and music generation modules
   //

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


   wire tempo_ready;
   wire tonality_ready;

   wire [31:0] tempo_period;
   wire tonality;
   wire ped_enable;
   wire beat;
   wire chord_ready;
   wire notes_ready;
   wire [4:0] chord;
   wire [4:0] cel_note;
   wire [4:0] vla_note;
   wire [4:0] vln2_note;
   wire [4:0] vln1_note;
   wire [31:0] beat_count;
   wire [1:0] rand;

   ped_filter pf0(.clock(clock_27mhz),.reset(reset),.noisy(user1[0]),
                  .clean(ped_enable));

   tempo_gen tempo_generator(.reset(reset),.clock(clock_27mhz),
                             .ped_enable(ped_enable),
                             .tempo_ready(tempo_ready),
                             .tempo_period(tempo_period));

   tonality_gen tonality_generator(.reset(reset),.clock(clock_27mhz),
                                   .tempo_ready(tempo_ready),
                                   .tempo_period(tempo_period),
                                   .tonality(tonality));

   beat_gen beat_generator(.reset(reset),.clock(clock_27mhz),
                           .tempo_ready(tempo_ready),
                           .tempo_period(tempo_period),.beat(beat));

   random random1(.clock(clock_27mhz),.reset(reset),.rand(rand));

   chord_generator chord_generator1(.clock(clock_27mhz),.reset(reset),
                                    .beat(beat),.tonality(tonality),
                                    .rand(rand),.chord(chord),
                                    .chord_ready(chord_ready));

   note_generator note_generator1(.clock(clock_27mhz),.reset(reset),
                                  .chord_ready(chord_ready),.chord(chord),
                                  .rand(rand),.cel_note(cel_note),
                                  .vla_note(vla_note),.vln2_note(vln2_note),
```

```
                                .vln1_note(vln1_note));

//////////////////////////////////////////////////////////////////////////
   //
   // Video output modules
   //
//////////////////////////////////////////////////////////////////////////

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

   wire phsync,pvsync,pblank;
   reg [7:0] red,green,blue;
   reg b,hs,vs;

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = red;
   assign vga_out_green = green;
   assign vga_out_blue = blue;
   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;

   // Synchronize signals from other sections
   wire trigger;
   ped_filter #(.DELAY(13000000)) pf1(.clock(clock_65mhz),.reset(reset),
                                      .noisy(user1[0]),.clean(trigger));

   wire [4:0] chord_v;
   synchronize #(.WIDTH(5)) s0(.clock(clock_65mhz),.in(chord),.out(chord_v));

   wire [31:0] tempo_period_v;
   synchronize #(.WIDTH(32)) s1(.clock(clock_65mhz),.in(tempo_period),
                                .out(tempo_period_v));

   // Visualization module
   wire [7:0] visual_r,visual_g,visual_b;
   visual v1(.vclock(clock_65mhz),.reset(reset),.hcount(hcount),
             .vcount(vcount),.hsync(hsync),.vsync(vsync),.blank(blank),
             .phsync(phsync),.pvsync(pvsync),.pblank(pblank),
             .trigger(trigger),.out_r(visual_r),.out_g(visual_g),
             .out_b(visual_b));

   // Music Info module
   wire [2:0] music_info_pixel;
   music_info minfo1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
                     .chord(chord_v),.tempo_period(tempo_period_v),
                     .mpixel(music_info_pixel));
```

```
   // switch[0] selects which video generator to use:
   //  0: Music Information
   //  1: Visualization
   always @(posedge clock_65mhz) begin
      if (switch[0] == 0) begin
         hs <= hsync;
         vs <= vsync;
         b  <= blank;
         red   <= {8{music_info_pixel[2]}};
         green <= {8{music_info_pixel[1]}};
         blue  <= {8{music_info_pixel[0]}};
      end
      else begin
         hs <= phsync;
         vs <= pvsync;
         b <= pblank;
         red <= visual_r;
         green <= visual_g;
         blue <= visual_b;
      end
   end



//////////////////////////////////////////////////////////////////////////
   //
   // Audio output modules
   //

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

   // allow user to adjust volume
   wire vup,vdown;
   reg old_vup,old_vdown;
   debounce bup(.reset(reset),.clock(clock_27mhz),.noisy(~button_up),
                .clean(vup));
   debounce bdown(.reset(reset),.clock(clock_27mhz),.noisy(~button_down),
                  .clean(vdown));
   reg [4:0] volume;
   always @ (posedge clock_27mhz) begin
      if (reset) volume <= 5'd8;
      else begin
         if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
         if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
      end
      old_vup <= vup;
      old_vdown <= vdown;
   end

   // AC97 driver
   wire signed [17:0] audio_out_data_low;
   wire signed [17:0] audio_out_data_high;
   wire signed [17:0] audio_out_data;
   wire received_audio, ready;

   stereo_audio a(.clock(clock_27mhz),.reset(reset),.volume(volume),
```

```
                .audio_out_data(audio_out_data),.ready(ready),
                .audio_reset_b(audio_reset_b),
                .ac97_sdata_out(ac97_sdata_out),
                .ac97_sdata_in(ac97_sdata_in),
                .ac97_synch(ac97_synch),.ac97_bit_clock(ac97_bit_clock));

   // High-level instrument modules
   wire vln1_ready;
   wire signed [15:0] vln1_out;
   violin vln1(.clock(clock_27mhz),.reset(reset),
               .received_audio(received_audio),.vln_note(vln1_note),
               .vln_ready(vln1_ready),.vln_out(vln1_out));

   wire vln2_ready;
   wire signed [15:0] vln2_out;
   violin vln2(.clock(clock_27mhz),.reset(reset),
               .received_audio(received_audio),.vln_note(vln2_note),
               .vln_ready(vln2_ready),.vln_out(vln2_out));

   wire vla_ready;
   wire signed [15:0] vla_out;
   viola vla(.clock(clock_27mhz),.reset(reset),
             .received_audio(received_audio),.vla_note(vla_note),
             .vla_ready(vla_ready),.vla_out(vla_out));


   wire cel_ready;
   wire signed [15:0] cel_out;
   cello cel(.clock(clock_27mhz),.reset(reset),
             .received_audio(received_audio),.cel_note(cel_note),
             .cel_ready(cel_ready),.cel_out(cel_out));

   // Mixer to produce data sent to AC97

   mixer mixer1(.clock(clock_27mhz),.reset(reset),.vln1_ready(vln1_ready),
                .vln2_ready(vln2_ready),.vla_ready(vla_ready),

.cel_ready(cel_ready),.vln1_out(vln1_out),.vln2_out(vln2_out),
                .vla_out(vla_out),.cel_out(cel_out),
                .audio_out_data(audio_out_data),
                .received_audio(received_audio));

endmodule
```