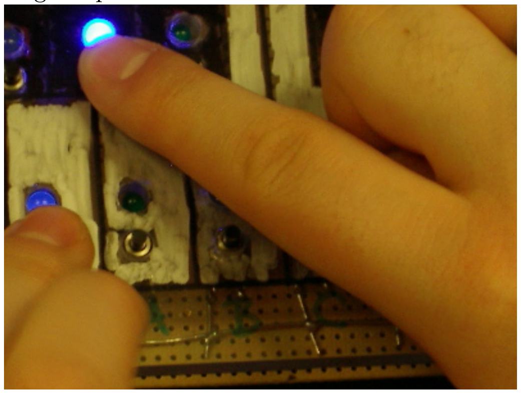
Digital Music Tutor: From Verilog to Virtuoso

Katherine H. Allen
Diane L. Christoforo

December 9, 2004

Abstract

This document outlines the design and implementation of the Digital Music Tutor: an electronic system to teach music using an interactive game. The modular components of the system allow independent testing and veri cation, while the I/O device allows the visual, in addition to aural, feedback. The system also has a "demonstration mode" which allows the user to watch the DMT play a given song, and a four-song ROM for variety. Although there were some small problems with the system in its nal con guration, it fulfills the design requirements and should be able to achieve its design goals.



1 Introduction: What is a Digital Music Tutor Anyway? (KHA)

Music is the universal language of mankind—every society has independently invented some sort of music. Rhythm and tones are built into the souls of human beings, and deeply embedded in our cultural heritage. Unfortunately, for most of us, perfect pitch and timing are *not* built in. We must be taught to feel rhythms, to sing or play an instrument, and to read music in whatever form it is presented. Repetition is the key to mastering any task, but mindless repetition gets tedious, and endless practicing often seems to have no payoff for music students. However, the same student presented with a video game will play the same sequence endlessly, each time improving until they get a high score—or beat their competitor. Therefore, if we can make practicing into a game, it stops being a tedious task and becomes an exciting challenge with immediate, as well as long-term payoff. The Digital Music Tutor combines a simple 12-tone keyboard input with audio and visual feedback and a scoring and feedback system to teach the user to play simple songs with precise timing.

2 Overview (KHA)

The DMT has two main parts: the Scoring Module (section 3.4), and the Sound Generator (section 3.5). The scoring module, in game mode, takes the user's input and scores it based on its resemblance to the song stored in the selected ROM. In demo mode, it sends enable commands to the sound generator as well as lighting the output device LEDs. The sound generator generates 12 possible tones and combines them for output to the speaker based on which tone is requested by either the output device or the Scoring Module.

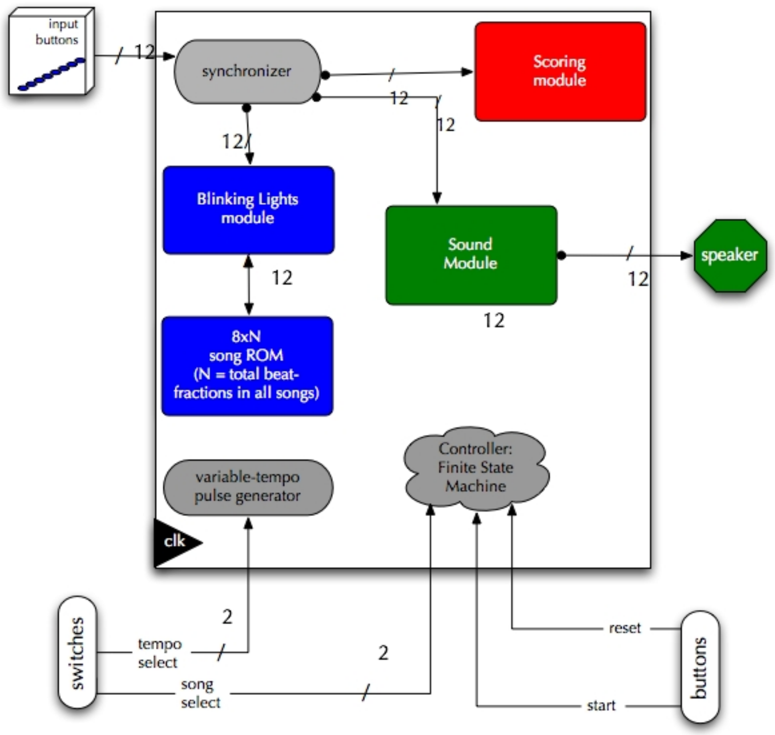


Figure 1: Module-level Block Diagram for the DMT

3 Detailed Design and Implementation

3.1 I/O devices (KHA)

3.1.1 Speaker

The speaker is connected to the output of an Analog Devices AD 558 Digital to Analog Converter. The converter is permanently enabled—the enables are shorted directly to ground, rather than being controlled by the FPGA. The DAC uses a range of 0 to 2.55v, with 256 potential levels. In practice, only 0 and 1.9-2.55 v outputs are commanded by the FPGA. (See section 3.5.4 for more details about the FPGA's commands to the DAC).

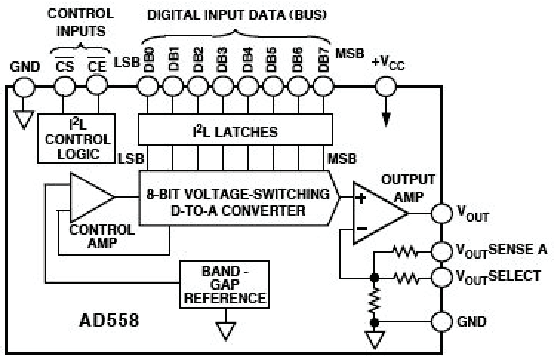


Figure 2: ADC 558 Functional Block Diagram

3.1.2 Piano-like Keyboard Controller

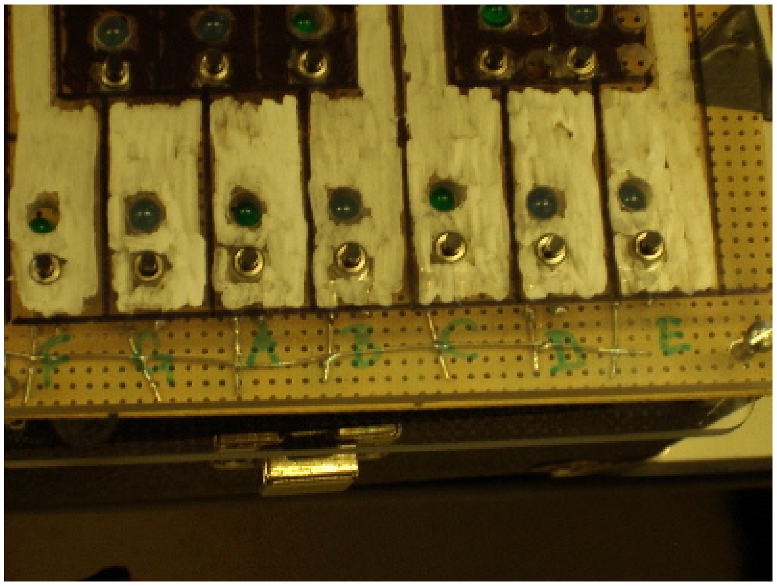
The Piano-like Keyboard Controller (PKC) is the input and output device for the Digital Music Tutor. Figure 3 shows the controller in its final form, from above and below.

The basic structure is two clear $\frac{1}{4}$" Plexiglas plates, separated 3.3 cm by standoffs. In the center is a piece of perforated circuit board with the buttons and LEDs. The 12 LEDs and push-button switches are set into a pattern resembling a piano keyboard over one octave, F# to E#. Each pushbutton switch has a #12 drill hole through the Plexiglas face, such that it fits perfectly into the hole. The buttons are secured with cyanoacrylate to prevent rotation. Each LED has a #9 hole into the Plexiglas, which fits the blue and red LEDs perfectly, but leaves some room for movement by the green replacement LEDs. The Plexiglas face is divided into thirteen sections—the seven white keys, five black keys, and the outer rim where the standoffs and the metronome light are located.

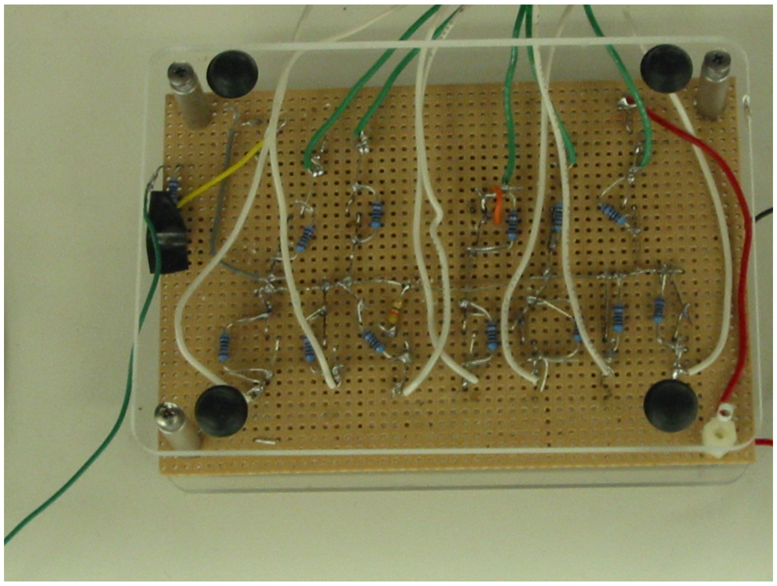
The circuitry of the PKC is relatively simple. Power (V_{cc}) is connected to the lab supply through a resistor, which reduces the input power to 4.5v. The power is connected to a bus which runs to all the button switches in parallel. The button switch output is tied to the positive terminal of the LED as well as the in/out line which runs to the FPGA. A 1000 ohm resistor is shorted across the LED as a pulldown, and the negative terminal of the LED is soldered to ground.

The pulldown is necessary in order to ground the circuit when the switch is open—without it, the voltage floats at 150mV, which is interpreted as a high voltage by the FPGA.

Originally, the PKC used twelve super-bright blue LEDs and one large red LED, but the voltage differential from the 5v power supply and the 0v ground in the labkit was too large. Five of the blue LEDs and one red LED shorted and had to be removed and replaced—the blue with smaller green LEDs, as no spare blue LEDs were available. The large red metronome LED (which blew up with a very impressive puff of smoke) was replaced with another red LED from the XVI electronics lab.



(a) Top View



(b) Bottom View

Figure 3: The Input Device ("PKC"): 12 notes, from F# to E#

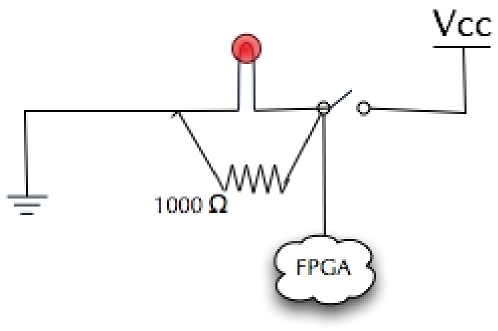


Figure 4: Circuit Diagram for One Note of the “PKC”

3.2 The Demo-And-Game FSM (Finite State Machine)

The actual gameplay is controlled by the *demo_and_game* FSM (finite state machine). Though the Digital Music Tutor contains a fair number of modules, the system's behavior is repetitive enough for a simple control system.

The system can be in demo mode or in game mode, controlled by the *demo_or_game* switch. The two modes perform nearly identically. There are two differences. First, in game node, the *accumulator_reset* has been turned off, so that the game is actually scored. Second, in demo mode the *enable* signal is turned on so that the lights and sounds play from the FPGA. In game mode, this ability is disabled so that the FPGA and the user do not try to light an LED simultaneously. (Simultaneous lighting would cause a short and possibly destroy the FPGA. While ”exploding lab kit,” may have been an interesting final project, the lack of recoverability made us choose someting less violent.)

Here, states have been given names corresponding to their actual function. The code still contains the now-completely-unrelated-to-function state names. At the beginning of each code file, a ”translation” has been placed explaining which wire/state/register/etc. name in the paper corresponds to the names in the code.

See Figure 5 for a picture of the conceptual FSM. The actual FSM consisted of two completely separate sets of states for demo mode and game mode. After reset, the system checks the *demo_or_game* switch, picks the appropriate ”trail,” and then follows it deterministically. In the figure, we have combined states with identical outputs and used *demo_or_game* to switch between the demo states and the game states at the point they actually branch off. (For example, the actual FSM has both a DEMO_INTRO and a GAME_INTRO state. We left our rather convoluted code alone as soon as it was working, but the figure shows a better view of what we intended.)

On a reset, the system moves into the RESET state. Then, the system moves into DEMO_START or GAME_START, depending on the condition of the *demo_or_game* switch. Here, the user can flip the song- select switches around. As soon as the player presses *start*, the system moves into the DEMO_ or GAME_INTRO state, and the metronome ticks out a measure's worth of beats as an introduction to the song. After that, the system moves directly into STARTING_SONG, which, in retrospect, is a completely superfluous state. Originally, it was in place to allow the memory address of the first line of the song to settle, but the INTRO state provides more than enough time for that, now. So, after one apparently-wasted clock cycle, we actually start playing the song, by going into DEMO_ or GAME_PLAYING_SONG. In DEMO_PLAYING_SONG, the *counter_reset* is disabled, and the lights are enabled. In GAME_PLAYING_SONG, the *counter_reset* is also disabled (we need to step through the memory or the song will not play...), and the *accumulator_reset* is also disabled. When the song ends, in demo mode we go to a single-tick ENDING_SONG state (in which we disable the lights and the counter again), and then back to start). In game mode, we loop in the ending state until start is pressed again. This way, the score remains displayed until

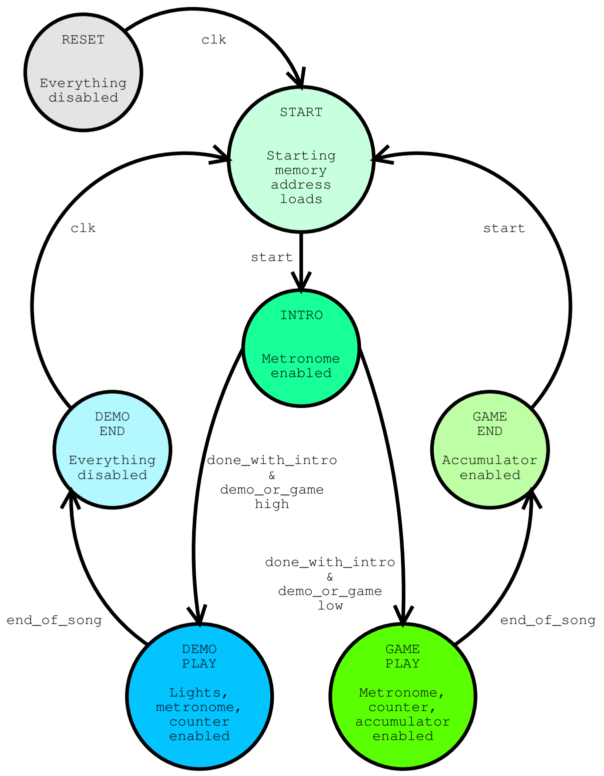

RESET
Everything disabled
clk
START
Starting memory address loads
clk
start
start
INTRO
Metronome enabled
DEMO END
Everything disabled
GAME END
Accumulator enabled
done_with_intro & demo_or_game high
done_with_intro & demo_or_game low
end_of_song
end_of_song
DEMO PLAY
Lights, metronome, counter enabled
GAME PLAY
Metronome, counter, accumulator enabled

Figure 5: Control Flow of the Finite State Machine

the user is ready to play another song.

3.3 Memory (DLC)

The song ROM holds four songs. The beginning of each song is at a location hardwired into the song selection switches, and the end is marked by setting all the notes to 1111 1111 1111. In this way, we allow songs to be of variable lengths. Within a song, each address indicates a row of 12 bits, which are high or low for "play this note" or "don't play this note", respectively. The ROM files are included in appendix C.2

3.3.1 ROM format

Each song begins with a line of all zeroes - 0000 0000 0000. As soon as the memory index points to the starting line of the chosen song, that line's output will fill all seven of the timing registers, and we want those to remain empty until we actually begin stepping through the notes of the song. Each note lasts for 3x lines, where x is an integer. (Since the metronome flashes on for three clock ticks and off for three clock ticks, having the notes last for multiples of three makes them match up with the beat nicely.) A line of all 1s - 1111 1111 1111 - in the *early1* register will send an *end-of-song* signal to the game-playing FSM.

The ROM has 1024 lines, allowing for four songs of variable length, although our sample ROM file has all songs 256 lines each. The ROM is unregistered. The address-counter, however, is registered off of the slow clock, so the ROM changes lines only once per slow-clock cycle.

3.4 Synchronization, Timing and Scoring (DLC)

3.4.1 Clock, Slow-Clock, and Metronome

The song-playing and -scoring part of the system uses three different clock signals. The normal clock is the 1.8MHz crystal oscillator we've all come to know and love. This clock runs the game-playing FSM. The slow clock, which ticks either 20 or 10 times per second, runs the scoring registers, the memory, the timing_checkers, and the accumulator.

The metronome is the player-clock. It turns on for three slow-clock cycles, then off for three slow-clock cycles, then repeats. This clock is slow enough for the player to use it to count out the tempo. All songs run in 3/4 time. A timing diagram for the clocks is included in figure 6.

3.4.2 The Scoring Setup

Since this project was inspired by DDR (the arcade and console video game Dance Dance Revolution, in which the player presses arrow buttons with their feet in time to music), our orignal goal was to keep our scoring identical to theirs. In DDR, the user is not penalized for holding down buttons when there is no step present - for standard notes, they are only scored when a note passes by the top "step on the button now" bar. Some arrows, known as "freeze" arrows, do require the user to press and hold the button for a particular length of time, and they are penalized if they let up too early. Finally, there are "jumps" - multiple arrows that occur at the same time. The user must make both correctly or they are penalized.

We actually did virtually the opposite. Instead of only checking the score when a note passes by, we only check the score when the user presses or releases a button. In addition, we score each button individually - the user can get a chord partially correct if they hit, say, one not early and one note on time. Also, all of our notes are treated as "freeze" notes - press and release times both matter. We do score like DDR in one way - the user gets a variable number of points depending on how close to the actual note they were.

The user's button presses first pass through a synchronizer (fast- clocked). Then the signals enter the *current button results* registers, and one slow-clock cycle later, the *previous button results* registers. Using these two registers, we can detect whether the user has just pressed a given button, just released, is holding down, or is not pressing at all.

We only adjust scores on a button press or release - if the user has a correctly timed press and release for a given note, we know she held the note down for the correct duration. (If the user completely misses -

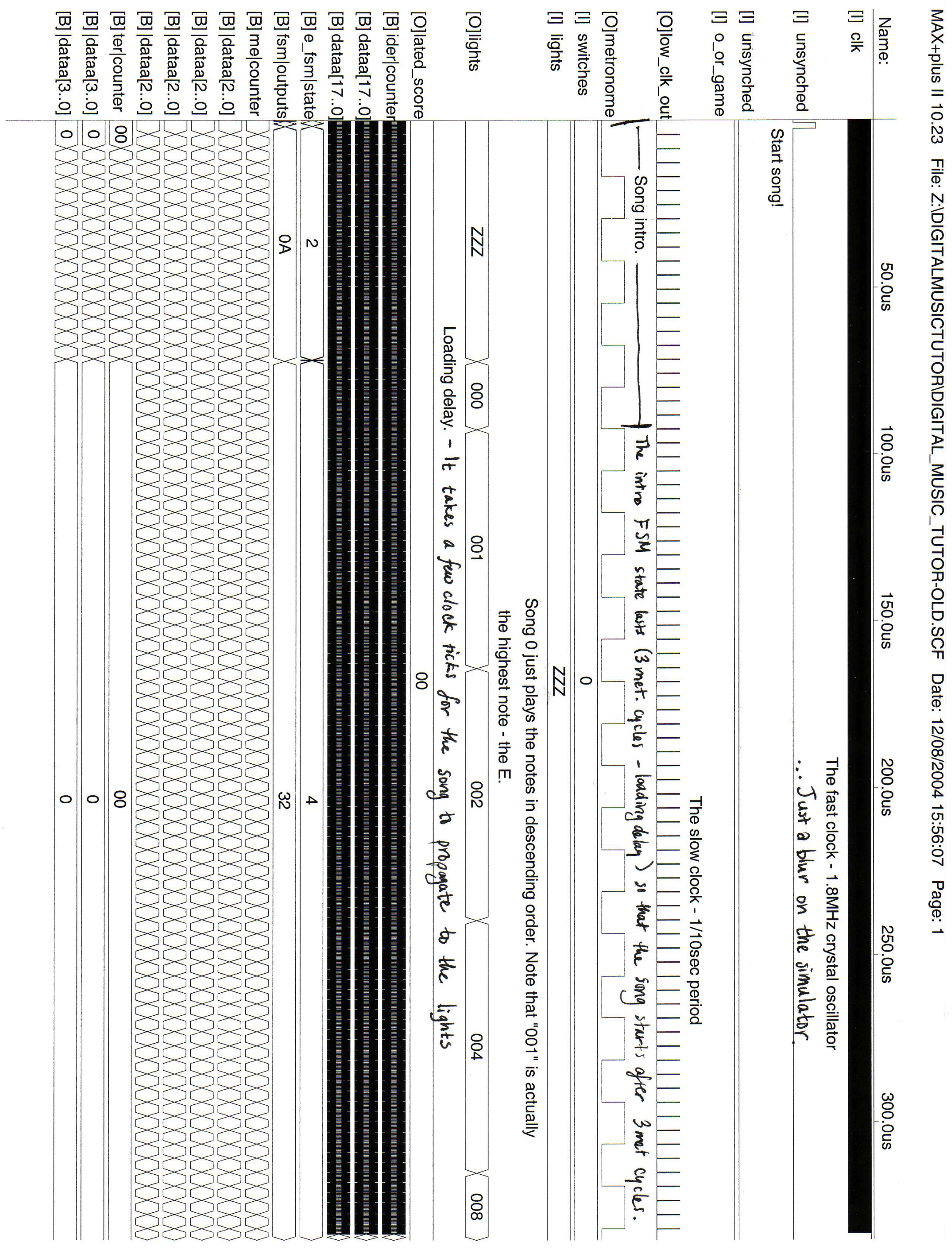


Figure 6: Timing of the Clock, Slow Clock and Metronome

fails to press OR release at any time during the entire span of a note, she is NOT penalized. See "Scoring Issues" for an explanation of why.)

After the note combinations for a particular slow-clock cycle come out of the song ROM, they flow through a series of seven slow-clocked registers. These registers are named *early1*, *early2*, *ontime1*, *ontime2*, *ontime3*, *late1*, and *late2*. They form a "window" around the notes playing at the current moment (the notes in *ontime2*). For each button, there is a *timing_checker* module which compares user input to the notes in the window.

3.4.3 On-Time, Early, Late, or Just Plain Wrong?

A button press is "on time" if the leading edge of the note occurs within one of the three *ontime* registers. That is to say, the signal is high in *ontime1*, *ontime2*, or *ontime3*, and *not* high in *late1*. By the same token, a release is "on time" if the trailing edge is contained within the *ontime* registers. We have three *ontime* registers rather than one to allow the user some leeway with their timing - given a slow-clock running at 10 ticks/second, the user has a 3/10 second window in which to be completely correct. A press/release scored as "on time" gives the user two points. (So the maximum score for any song is 4*number of notes.)

If the press/release is not on time, then the checker looks for an "almost." A press is an "almost" if the starting point of the note occurs at *late1*, and a release is "almost" if the ending point of the note occurs at *early2*. An "almost" is worth one point.

If the user presses or releases on any condition *other* than an "on time" or an "almost," then one point is subtracted from their score. The user can make his score go arbitrarily negative (within the range of what can be represented in eight bits) by mashing buttons repeatedly. (That strategy may work well in fighting games, but timing matters here.)

If the button is held or unpressed, then no score change happens. This design can lead to the amusing situation where the user presses down at the beginning of one note and does not release until the end of the *next* note. Both the press and release are scored as "correct." The user does not lose the points an incorrect release-and-press would have cost her, but neither does she gain the points a correct release-and-press would have gotten.

3.4.4 The Final Score

After each slow-clock tick, the twelve timing-checkers output the score for each note to the accumulator, which sums them. Note that each timing-checker sends two bits of output to the accumulator - 0, 1, 2, or 3 - while we want scores of -1, 0, 1, or 2. The accumulator adjusts by subtracting 12 from the score each slow-clock tick. The final score is output in 8-bit two's complement binary to both the 8 lab kit LEDs and to two squares on the FPGA hex lights.

3.5 Sound generation (KHA)

3.5.1 How it works

The sound generator has 12 "tones", which are square waves at between 349 Hz and 659 Hz, corresponding to the octave from the F above middle C to the E at the top of the treble clef. These are then digitally combined and sent as a value between 0 and 255 to the DAC, which outputs a voltage between 0 and 2.55v to the speaker.

3.5.2 Selected tones

The particular tones were selected to be easy to display on a five line treble clef—they correspond to the lowest "space" through the highest "line" (see figure 9). Since we chose not to implement the video display, however, this is less relevant than it might have been—any 12 tones could have been implemented with minimal changes. The particular frequencies correspond to the Equal-Tempered Tuning, which is

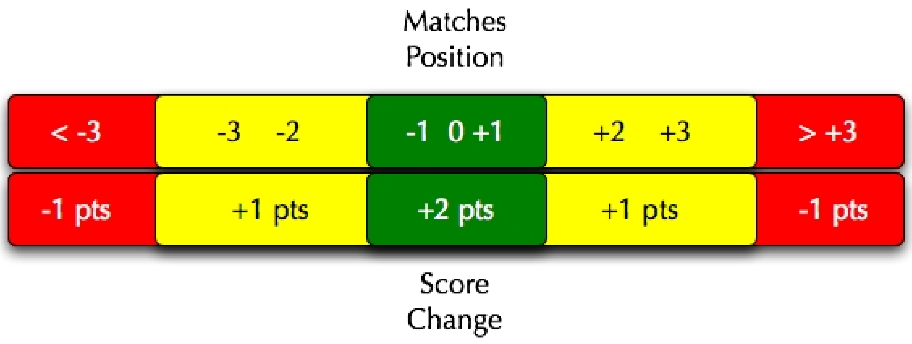


Figure 7: Score and Visual Indication for Correct Input

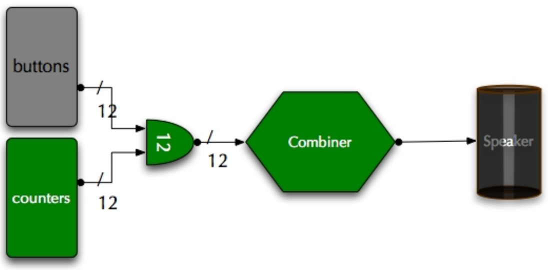


Figure 8: Block Diagram for the Sound Generator

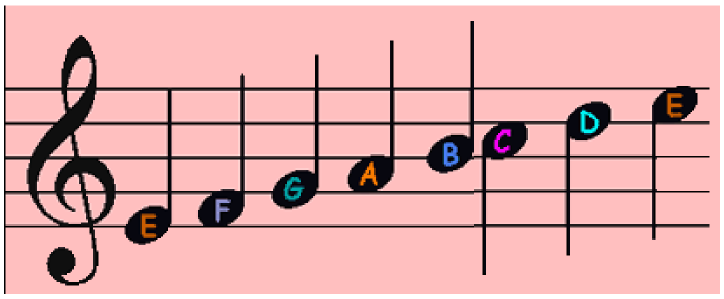


Figure 9: Notes Playable on the DMT

Table 1: Equal-Tempered Tuning for the Twelve Tone Scale

Note Name	**Frequency (Hz)**	**1.8432 Mhz Clock Ticks per Cycle**
F	349.228	5278
F#	369.994	4982
G	391.995	4702
G#	415.305	4438
A	440	4189
A#	466.164	3990
B	493.883	3732
C	523.251	3522
C#	554.365	3324
D	587.33	3138
D#	622.254	2962
E	659.255	2795

mathematically calculated such that songs sound "correct" when transposed into any key. Table 1 shows the tones and the clock dividers required to implement them.

3.5.3 Why Square Waves?

Our original intention had been to use 32-bit sampled sine waves, stored in a ROM and played at variable rates. However, we discovered that a sine wave has only marginally more aesthetic quality than a square wave. Square waves have the advantage of being extremely simple to generate, and are easier to digitally combine than ROM samples.

In our sound module, tones are generated using a generic clock divider (clkdiv.v) which counts a certain number of clock cycles before flipping from 1 to 0 or vice-versa. Each of the twelve tones is generated independently with a separate instantiation of this clock divider. This means that the signals do not have to be synchronized, there is minimal delay, and the design is elegantly simple.

3.5.4 Sound combination

Playing multiple tones simultaneously was significantly more of a challenge than playing a single tone. However, the selected implementation is simple and elegant, requiring very little Verilog. In order to digitally combine the signals, the FPGA adds the twelve signals and converts the result to an eight-bit digital value based on how many of the tones are high. The result is sent to the DAC, where it is converted to an analog voltage between 0 and 2.55v and sent to the speaker.

The resulting tone is noisy, but one, two or three tones can be played with appropriate resulting chords. Four or more tones played concurrently sounds increasingly like random static, as the competing waves keep the DAC at its maximum value for more and more of its cycle, making it impossible to notice the various frequency tones. Section 4.2 discusses the multi-tone noise in detail.

4 Issues

4.1 Scoring Sensitivity (DLC)

"This game teaches you either to play music or to press buttons really fast."

Originally, we only had a single register to hold each button's current value, which meant we could not distinguish a press from a release and thus had to be more lenient with scoring than we wanted to be. This problem was fixed with the addition of a second register, so that we could catch the slow-clock cycle on which the button went from 0 to 1 or 1 to 0.

An issue we did not have a particularly satisfactory solution for concerned "missing" notes - where the user fails to press at any time during a particular note's duration. In the first version of the scoring code, if all three *ontime* registers were 1 and the user was neither pressing, holding, nor releasing, then a "miss" was scored. Clearly, there was a note in progress, and the player had dropped the ball, so to speak. However, in practice, this caused the score to *race* negative. While the concept of "more penalties the longer you miss," was nice, this setup made it impossible to get a non-negative score in the game.

We also might have been able to do the DDR style "a miss is scored if the user has failed to press the button by the time the note edge passes the *late* point." Scoring in this method would have required nine registers - a third late and early - so that we could identify the point after "late" for a press and before "early" for a release. In retrospect, I think we failed to implement this solely because we spent so much time working on other aspects - it would not be a huge code change to implement this.

The game is also *hard*. Because we did not implement video, the player is completely dependent upon the metronome (and their memory) for telling when to press a note. If the user loses the beat, they will probably never find it again. So by *not* continually penalizing missed notes, the user can play to see how long he can go without losing the tempo, and he'll still have his last score at the song's end. The maximum score thus far achieved is 28—see figure 10.

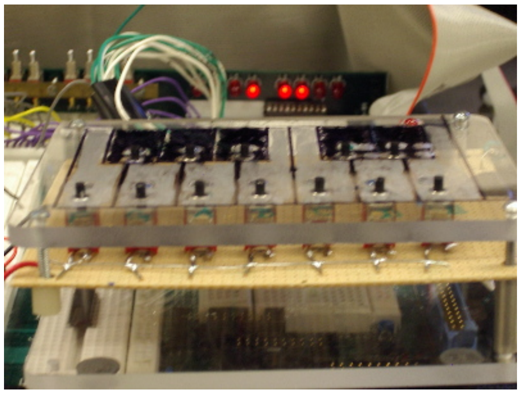


Figure 10: Diane's High Score: 28

4.2 Multi-tone noise (KHA)

As mentioned in section 3.5.4, the sound combiner has a significant amount of high and low frequency noise caused by the overlap of square waves for a majority of their periods. The result is a signal that is high most of the time, with occasional glitches to low, which cause a "ticking" sound. A sine wave would have somewhat less of this "ticking", as more levels of sampling could be transmitted, but a perfectly reconstructed combination of sine waves would require either complicated software with very large samples or multiple DACs.

This is generally not a problem for "real" instruments since they use rich combinations of overtones at multiples of the original frequency, superimposed in analog-land, not digital. To improve the quality of the sound, we could use sampled and linearly-interpolated sinusoids stored in a ROM. Additionally, we could potentially create overtones (either for square waves or sine waves) at appropriate frequencies such that the keyboard tones sound less "electronic" and combine more aesthetically.

5 Conclusion (KHA)

Despite some minor problems, the system achieves its design goals, and is, in general, a very entertaining and addictive game. There is plenty of potential room for expansion, in particular adding video to the input, and perhaps increasing the size of the buttons. Although we will probably not be the next DDR, it was certainly entertaining to design and build, and satisfying to complete.

A Synthesizer (KHA)

A.1 Clock Divider

```
module clkdiv(clk,q, reset);}
   input clk, reset;}
   output q;}
   parameter BITS=32;}
   //maxcount should be 1/2 the number of clock ticks per period that you want.}
   //i.e. setting maxcount to 1 will give you a clock 1/2 as fast}
   //as the clock being fed into the module}
   }
   parameter maxcount = 20; // default is small for testing purposes}
 }
   reg [BITS−1:0] count;}
   reg             q_reg;}
 }
   always @(posedge clk)}
     begin}
        if(reset)}
          count <= 0;}
        else //!reset}
          if (count < maxcount)}
            begin}
               count <= count + 1;}
               q_reg <= q_reg;}
            end}
          else}
            begin}
               count <= 0;}
               q_reg <= ~q_reg; // Flip the bits}
            end}
     end // always @ (posedge clk)}
   assign q = q_reg;}
endmodule}
```

A.2 Sound Generation

```
module soundmodule(clk, reset, enables, tones, speakeroutput);}
//module soundmodule(clk, reset, enables, tones);//for running w/o soundcombiner}
   input clk, reset;}
   input [11:0] enables;}
   output [11:0] tones; // each bit is one of the 12 tones}
   reg [11:0]    tones;}
   output [7:0]  speakeroutput; // the combined tones out to DAC and speaker}
   }
   sound_combiner soundcombiner(clk, reset, tones, speakeroutput);}
   }
   clkdiv Fnat(clk, Fnatclk, reset);}
   defparam   Fnat.BITS = 13;}
   defparam     Fnat.maxcount = 5278;}
//   defparam  Fnat.maxcount = 4; // for testing}
```

```
    }
    clkdiv Fshp(clk, Fshpclk, reset);}
    defparam    Fshp.BITS = 13;}
    defparam      Fshp.maxcount = 4982;}
    //defparam  Fshp.maxcount = 2; // for testing}
}
    clkdiv Gnat(clk, Gnatclk, reset);}
    defparam    Gnat.BITS = 13;}
    defparam      Gnat.maxcount = 4702;}
    //defparam  Gnat.maxcount = 3; // for testing}
    }
    clkdiv Gshp(clk, Gshpclk, reset);}
    defparam    Gshp.BITS = 13;}
    defparam      Gshp.maxcount = 4438;}
    //defparam  Gshp.maxcount = 13; // for testing}
    }
    clkdiv Anat(clk, Anatclk, reset);}
    defparam    Anat.BITS = 13;}
    defparam      Anat.maxcount = 4189;}
    //defparam  Anat.maxcount = 4; // for testing}
    }
    clkdiv  Ashp(clk, Ashpclk, reset);}
    defparam    Ashp.BITS = 12;}
    defparam      Ashp.maxcount = 3990;}
    //defparam  Ashp.maxcount = 11; // for testing}
    }
    clkdiv Bnat(clk, Bnatclk, reset);}
    defparam    Bnat.BITS = 12;}
    defparam      Bnat.maxcount = 3732;}
    //defparam  Bnat.maxcount = 10; // for testing        }
}
    clkdiv Cnat(clk, Cnatclk, reset);}
    defparam    Cnat.BITS = 12;}
    defparam      Cnat.maxcount = 3522;}
    //defparam  Cnat.maxcount = 5; // for testing}
    }
    clkdiv Cshp(clk, Cshpclk, reset);}
    defparam    Cshp.BITS = 12;}
    defparam      Cshp.maxcount = 3324;}
    //defparam  Cshp.maxcount = 6; // for testing}
    }
    clkdiv Dnat(clk, Dnatclk, reset);}
    defparam    Dnat.BITS = 12;}
    defparam      Dnat.maxcount = 3138;}
    //defparam  Dnat.maxcount = 7; // for testing}
    }
    clkdiv Dshp(clk, Dshpclk, reset);}
    defparam    Dshp.BITS = 12;}
    defparam      Dshp.maxcount = 2962;}
    //defparam  Dshp.maxcount = 8; // for testing}
    }
```

```
   clkdiv Enat(clk, Enatclk, reset);}
   defparam    Enat.BITS = 12;}
   defparam      Enat.maxcount = 2795;}
   //defparam  Enat.maxcount = 9; // for testing}
   }
always@(posedge clk)}
  begin}
    if(Fnatclk & enables[11])}
      tones[11] <= Fnatclk;}
    else}
      tones[11] <= 0;}
    }
    if(Fshpclk & enables[10])}
      tones[10]<=Fshpclk;}
    else tones[10] <= 0;}
     }
    if(Gnatclk & enables[9])}
      tones[9] <= Gnatclk;}
    else tones[9] <= 0;}
    }
    if(Gshpclk & enables[8])}
      tones[8]<=Gshpclk;}
    else tones[8] <=0;}
    }
    if(Anatclk & enables[7])}
      tones[7]<=Anatclk;}
    else tones[7] <= 0;}
    }
    if(Ashpclk & enables[6])}
      tones[6]<=Ashpclk;}
    else tones[6] <= 0;}
    }
    if(Bnatclk & enables[5])}
      tones[5]<=Bnatclk;}
    else tones[5] <= 0;}
    }
    if(Cnatclk & enables[4])}
      tones[4]<=Cnatclk;}
    else tones[4] <= 0;}
    }
    if(Cshpclk & enables[3])}
      tones[3]<=Cshpclk;}
    else tones[3] <= 0;}
    }
    if(Dnatclk & enables[2])}
      tones[2]<=Dnatclk;}
    else tones[2] <= 0;}
    }
    if(Dshpclk & enables[1])}
      tones[1]<=Dshpclk;}
    else tones[1] <= 0;}
```

```
      }
      if(Enatclk & enables[0])}
        tones[0]<=Enatclk;}
      else tones[0] <= 0;}
      }
   end // always@ (posedge clk)}
endmodule   }
```

A.3 Tone Combination

```
module sound_combiner(clk, reset, tones, d_out);}
   }
   input clk, reset;}
   input [11:0] tones;//from soundgenerator.v}
   output[7:0]  d_out; // to 8−bit DAC}
   reg [7:0]     d_out;}
   }
}
//Count up the number of ones in the tones signal}
// 11 ones is 11111111}
//  0 ones should be the minimum DAC value (00000000)}
   reg [3:0]     accum;}
   }
   always@(posedge clk)}
     begin }
        if(reset)}
          accum <= 0;}
        else     }
          begin }
                 //Count the number of ones coming in}
             accum <= tones[11] + tones[10]+tones[9]+tones[8]+tones[7]}
                +tones[6]+tones[5]+tones[4]+tones[3]+tones[2]+tones[1]+tones[0];}
             case(accum)}
               0: d_out <= 0;}
               1: d_out <= 160;}
               2: d_out <= 170;}
               3: d_out <= 180;}
               4: d_out <= 190;}
               5: d_out <= 200;}
               6: d_out <= 210;}
               7: d_out <= 220;}
               8: d_out <= 230;}
               9: d_out <= 240;}
               10: d_out <= 250;}
               11: d_out <= 255;}
             endcase // caseaccum}
          end // else: !if(reset)}
     end // always@ {posedge}
   }
endmodule}
```

B Timing, Scoring and Playback (DLC)

For the following sections of code, please reference the attached files. Our apologies for the inconvenience.

B.1 Control

B.1.1 Top Level

digital_music_tutor.v

B.1.2 FSM

demo_game_fsm.v

B.2 Timing

B.2.1 Tempo Divider

tempo_divider.v

B.2.2 Metronome

metronome.v

B.3 Scoring

B.3.1 Timing Checker

timing_checker.v

B.3.2 Score Accumulator

score_accumulator.v

C Song ROM

C.1 Song Start Translator

song_start_translator.v

C.2 Song ROM Demo File

songdemos.mif