ECSE 323 Digital System Design

Lab #5 - System Integration for the Music Box System

Group 27

Lulan Shen (260449509)
Loren Lugosch (260404057)

Part 1: Functional Description of the System

The Fall 2013 Digital System Design project was a music box made out of low-level digital hardware components programmed in an FPGA on an Altera educational circuit board with audiovisual inputs and outputs. In lab 5, we combine all of the components we made in the previous labs and make a few extra ones for connection logic and system display. Since someone using our music box needs to know how to configure the music as he/she wants, lab 5 is an exercise in UI/UX as well as system integration.

The music box plays a song as soon as it turns on, but it can be made to play again by pushing the restart button (the pushbutton marked “2”) or by putting the circuit in looping mode (the slide switch marked “0”), in which case it will keep playing the same song over and over. The circuit plays one of two songs (switching between them using the switch marked “1”) in a style controlled by the user’s selection of tempo and sound quality. The user can change the tempo using the seven leftmost slide switches, and the 2nd switch controls the sound quality by selecting one of two shapes for the musical waveform.

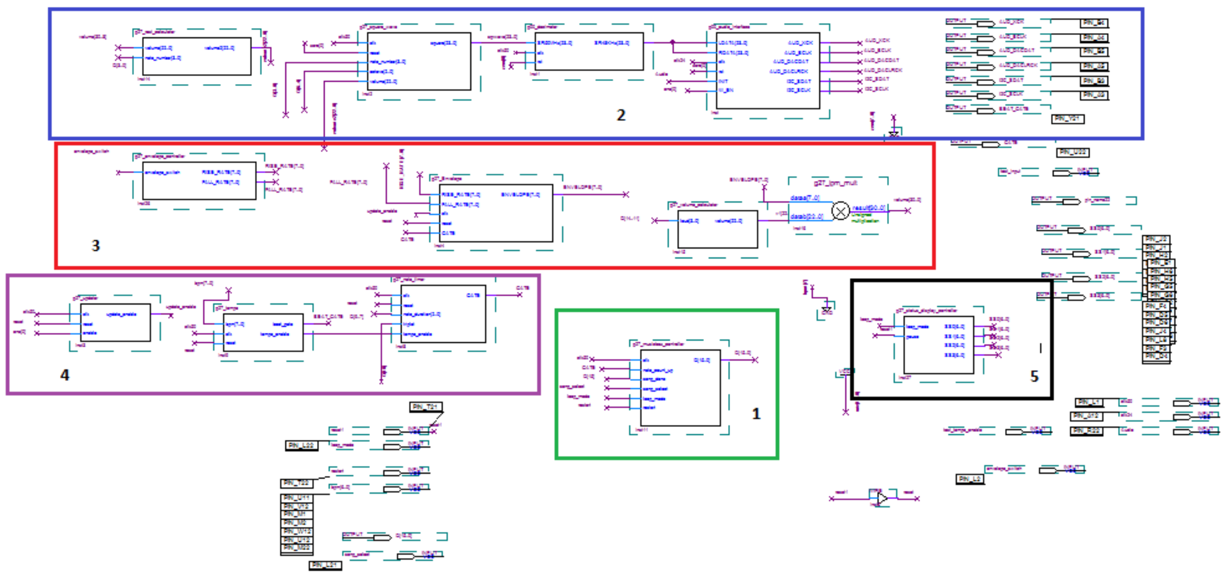


Figure 1 - Modular view of our complete system

The hardware components of the system can be categorized as belonging to one of 5 modules. Module 1 is the controller. It contains a finite state machine and two ROMs containing the notes of two songs, which act as instructions.

Modules 2 through 5 constitute the datapath. The datapath is manipulated by the bits of the notes outputted by the controller into playing the notes of the song in the right order and for the right time, and it feeds control bits back into the controller. Each module of the datapath plays a different role in shaping the output of the music box.

In Part 1 we review each of these modules and their relation to instructions.

1. Controller module (green)

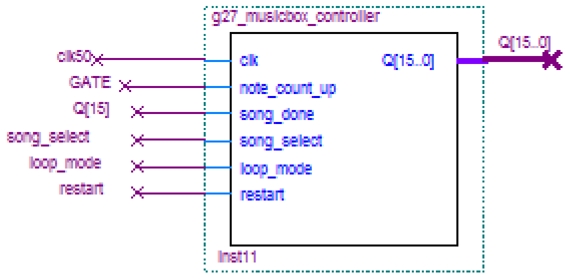
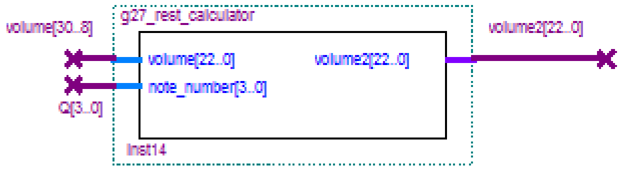


Figure 2 - Block diagram for FSM and song ROM

This module constitutes the circuitry which controls the overall operation of the system. We make a counter to keep track of which note is being played. This serve as the address for the song ROM. The controller should monitor the GATE output of g27_note_timer module to determine when the note is completed and then increment the address counter of the song ROM, to go to the next note. Q[15] is the end of song marker. Song_select input is connected to a slide switch to select a song to play. Restart input is connected to a pushbutton press to playback from the beginning of the song. Also playback can be in either a One-Shot mode or a Continuous Looping mode, selectable with a slide switch.

2. Sound generation module (blue)



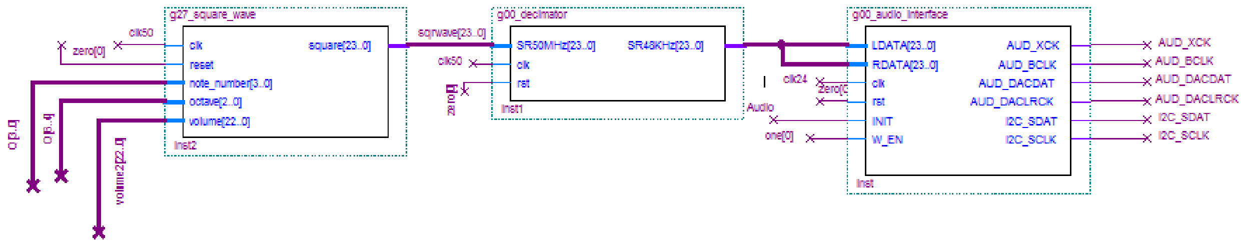


Figure 3 - Module 2 block diagram

This module creates the square wave of the appropriate frequency and plays it. This tone gets shaped by other modules in the music box, but without it, no sound comes out at all.

G27_rest_calculator
This module creates a lull in the music. The chromatic scale we use only has 12 notes, so bits 12 to 15 of note number are undefined. We set the volume to 0 when the note number is any number between 12 and 15. This gives us the ability to play music with rests. The rests have the same length as the corresponding tone-notes.

G27_square_wave
The music box tune to be played is described by a set of 16-bit note information words, which is the output Q of our music controller stored in the song memory. Q[3..0] represents the note number, indicating the note in the musical scale. Q[6..4] represents the octave number, indicating the octave the note should be played at. Increasing 1 in the octave number doubles the frequency of the note being played. We generate a basic waveform to provide that musical notes for each note number. The music box will use the square waveform.

G00_decimator
“This module converts the high sampling rate of 50MHz provided by our square wave generator to the 48kHz rate used by the codec chip. It also performs anti-aliasing filter to reduce aliasing noise.” - taken from lab description

G00_audio_interface
“This module implements the serial communication to the Wolfson WM8731 Audio Codec chip located on the Altera board. It also initializes various settings for the codec chip when the INIT line goes high.”- taken from lab description

3. Sound quality module (red)

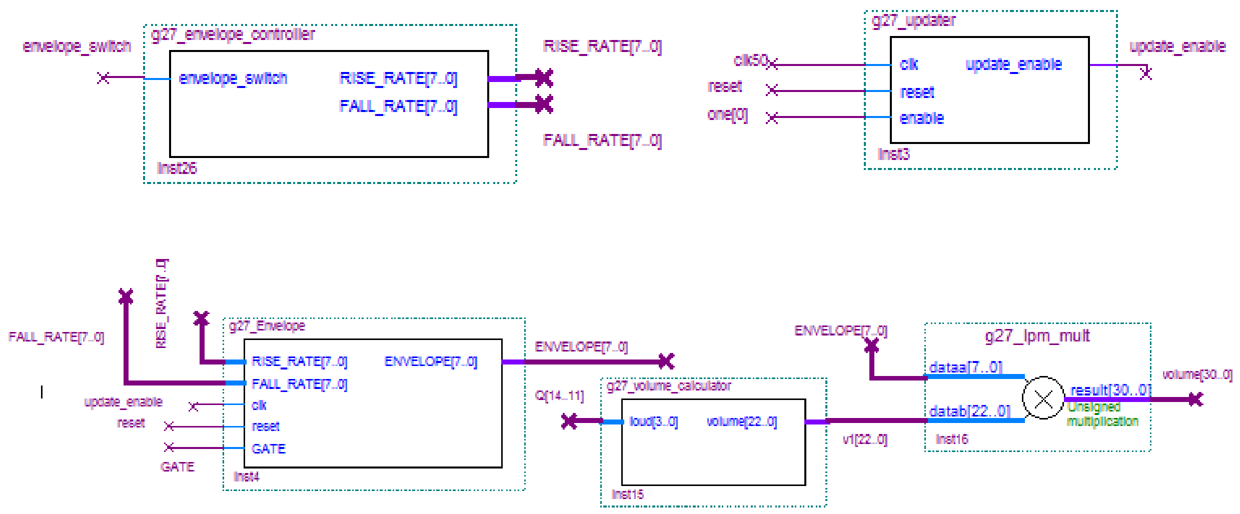


Figure 4 - Module 3 Block Diagram

The sound quality module makes the square wave more pleasant by multiplying it by various exponential signals.

G27_envelope_controller
This module is used to change envelope rates by envelope_switch which is connected one of the slide switches on the DE1 board.

G27_Envelope
We reshape the square wave so that the sound is more interesting. FALL_RATE and RISE_RATE input determine the slope of the shape by adding the last value of the envelope to the RISE/FALL_RATE every rising clock edge when UPDATE_ENABLE is high also. Gate signal controls the progression of the envelope signal. When GATE is high, envelope's value rises, and when GATE is low, envelope's value falls.

G27_updater
The module is used to create UPDATE_ENABLE input for g27_envelope module. UPDATE_ENABLE is a repetitive waveform that is high for one clock cycle and has 48kHz (which is 50MHz/1024) pulse rate.

G27_volume_calculator & g27_lpm_mult
Q[14..11] represents the volume intensity. The loudness setting in the note memory is converted to the 23 bit volume input and multiplied by the ENVELOPE output. Then we put the product (discard the first 8 bits) into the Volume input of g27_square_wave module.

4. Music logic / timing module (purple)

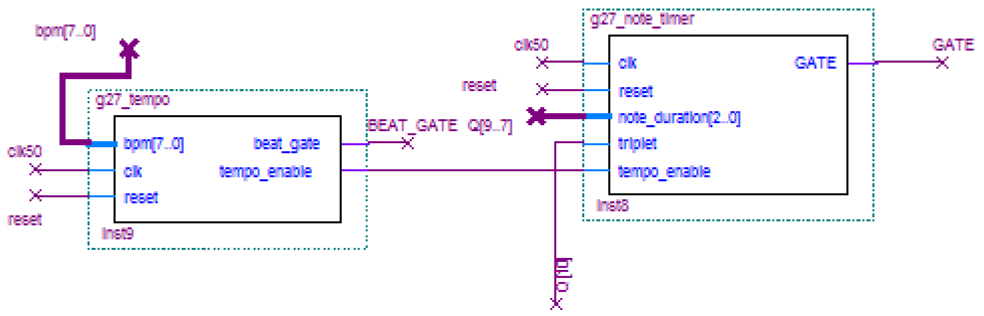


Figure 5 - Module 4 block diagram

This module (which might also be considered part of the controller) tells the circuit when to switch notes. The timing allows the circuit to play songs rather than single notes. This module is unique in that it partly controls the main controller by telling it (via GATE) when a note has finished playing so that the FSM can load a new note.

G27_tempo
This circuit has 24 pulses for each beat for setting basic tempo for music. The tempo in electric music instruments is conveyed in beats per minutes. Each beat is divided into subdivisions, allowing multiple notes in one beat. We adjust our tempo so that our music box can play both fast and slow songs.

G27_note_timer
Q[9..7] represents note duration, indicating how long the note will play. This is given in terms of a beat time fraction, which is determined by the overall tempo. Q[10] represents the triplet bit, indicating the note is 2/3 of a regular length note duration. This allows three notes to be played in the time of two notes. We designed the note_timer circuit that feeds in the duration of the note and triplet information, then generate a note gate signal, which is high for the suitable time.

5. User interface module (black)

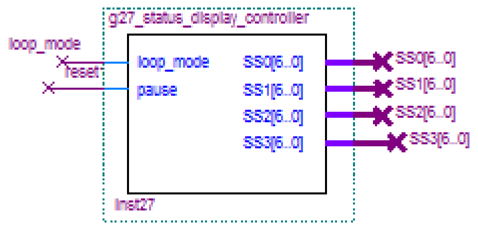


Figure 6 - Module 5 block diagram

This module indicates to the user whether the system is paused (the 7-segment display lights up like a “P”) and/or looping (the 7-segment display lights up like a “L”). Two of the 7-segment displays are left blank and could be used to give the user more feedback about the state of the music box. The circuit is composed of four multiplexers which select either a pattern of ones and zeros that will light up the LEDs correctly, or all ones to keep the active-high LEDs blank.

Part 2: Design of the System

Finite State Machine

The lab description mostly tells us how to hook the various bits of a song word up to components of the circuit, so we follow all of those suggestions. That gives us all we need for the circuit to work “statically” (i.e. playing a single tone for the right amount of time), but to get the circuit to work “dynamically” (switching notes to play a complete song) we need to do a little engineering.

We need a circuit which can load notes from a song ROM and play the notes for the correct amount of time with the correct tone at the correct octave. Thus our circuit needs at least one state with one output: playing a note, and a 16-bit word Q representing the note information.

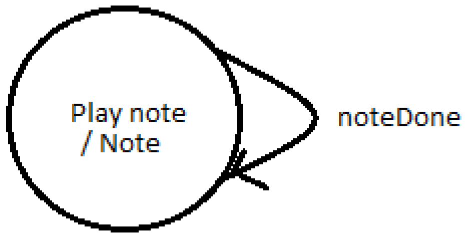


Figure 7 - Simple one-state Moore FSM

The dynamic nature of the circuit means that the circuit needs not only just to play a note, but to play each of the notes of the song once. To remember which note is being played so that the circuit knows which note of the song to play next and what the current controls to the various components of the datapath should be, we could create a massive 256-state Moore-type finite state machine in which the output of state i is the address of the location of the ith note of the song (as was our first idea):

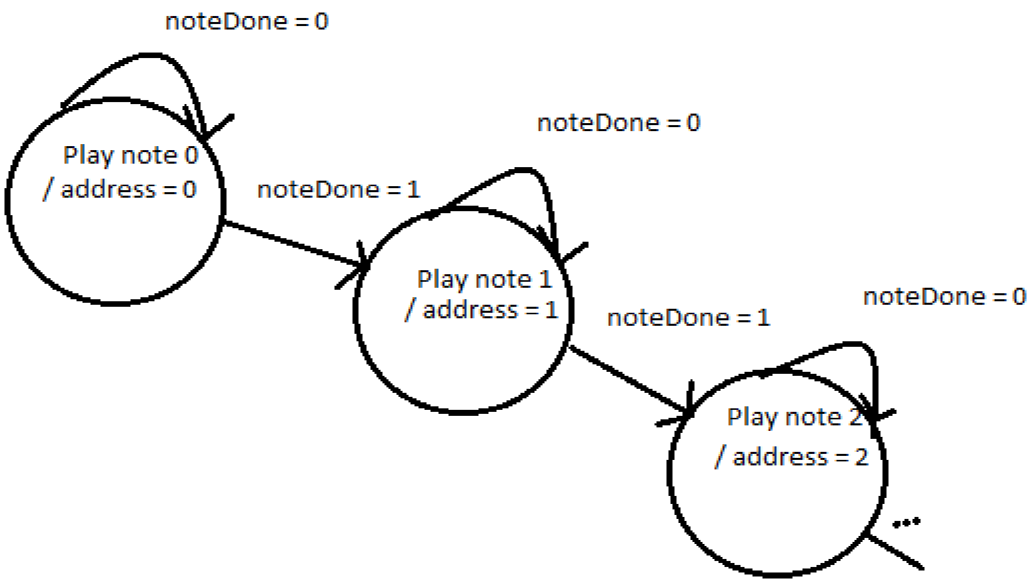


Figure 8 - 256 state machine

However, this design takes too long to write in VHDL and doesn't scale well (suppose that the song ROM were upgraded to 512 words or shrunk to 128 words; in those cases a huge amount of code would have to be rewritten if each state were handled within a case-statement inside a process block). We therefore use a Mealy machine in which the output is an increment to the counter and transitions either have this increment (if noteDone) or don't (if !noteDone). The address of the next note in the ROM is controlled by a counter ("address") which is incremented by state transitions. (A register with 8 bits essentially IS a flip-flop set for a 256-state FSM, but the hardware description for a two-state FSM plus a counter is much simpler because we don't need to think of counters as FSMs.)

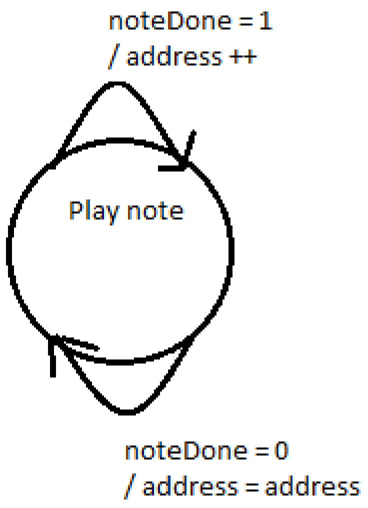


Figure 9 - FSM with counter

This circuit does not yet provide some of the functionality we want, though, like stopping the circuit's playing once the "end-of-song marker" is found. We could handle this case by simply switching the volume to 0 whenever Q[15] = 1, but that assumes that the last note and every note past it in the ROM has Q[15] = 1, which is not the case. We could also try using combinational logic external to the FSM which forces the circuit to set volume = 0 when the address is greater than the index of the first note with an end-of-song marker. However, this requires a preprocessing period in which the circuit loops through the song ROM, finds the last note, latches the index, and then plays the song, and that involves yet another FSM (with states like "preprocessing" and "playing"), which we want to avoid. Both of those approaches are susceptible to overflow problems (depending on the counter, when the count overflows, the song will restart whether we want it to or not).

Whenever the FSM is in state "Play note", there is a chance that it will increment the ROM address counter and keep the circuit playing. Therefore, we need to sequester the case in which end of song is reached and only jump back to "Play note" when the circuit has a new note to play (state "new note"). In our circuit, those states have names loadingNewNote and waitForNextNote.

However, we need to clarify what "note done" means in the first place. How does the circuit know that the note is finished? The note timer circuit plays a new note (rhythmically, not tonically) when the appropriate number of tempo enables have passed, but how do we transfer that knowledge to the FSM? Rather than trying to time the notes outside of the note timer, our circuit takes advantage of the fact that every note's GATE output looks like this:

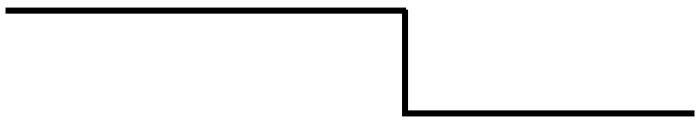


Figure 10 - GATE waveform for one note

The note is "done" when the next rising edge comes. Whenever GATE = 0, we load a new note and transition to the waiting state. So that the circuit doesn't increment multiple times during this time, we remain in the waiting state until GATE goes high again, then transition to the loading state. In the loading state, we wait for GATE = 0 to load a new note, and so on. Thus a new note gets loaded only at the falling edge of GATE, and each note is allotted the full GATE cycle.

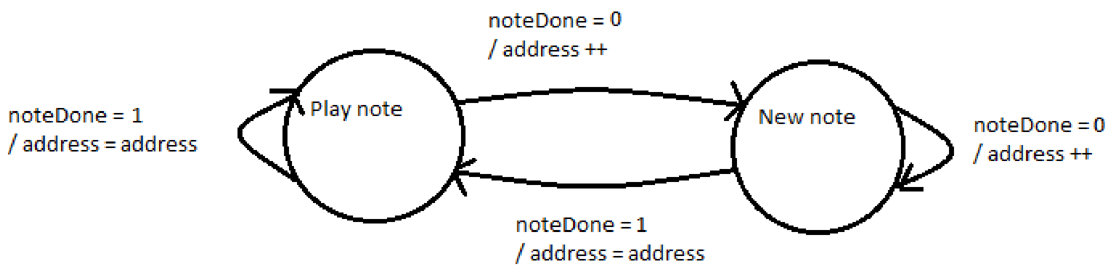


Figure 11 - FSM with loading/waiting distinction

Adding end-of-song checking to this circuit is a simple matter of remaining in the “new note” state (the state in which the circuit cannot possibly increment the song ROM address). How do we add loop mode? Our circuit is already required to have a restart input, so why not use that to set the ROM address to 0? We can’t do this because VHDL only allows one signal to drive a wire, so we can’t have both a button on the circuitboard and the FSM driving the restart signal. We can, however, transition to a state in multiple ways. All we have to do is add a “first note” state which occurs either when 1. the user pushes the restart button or 2. the circuit reaches end-of-song and is in loop mode.

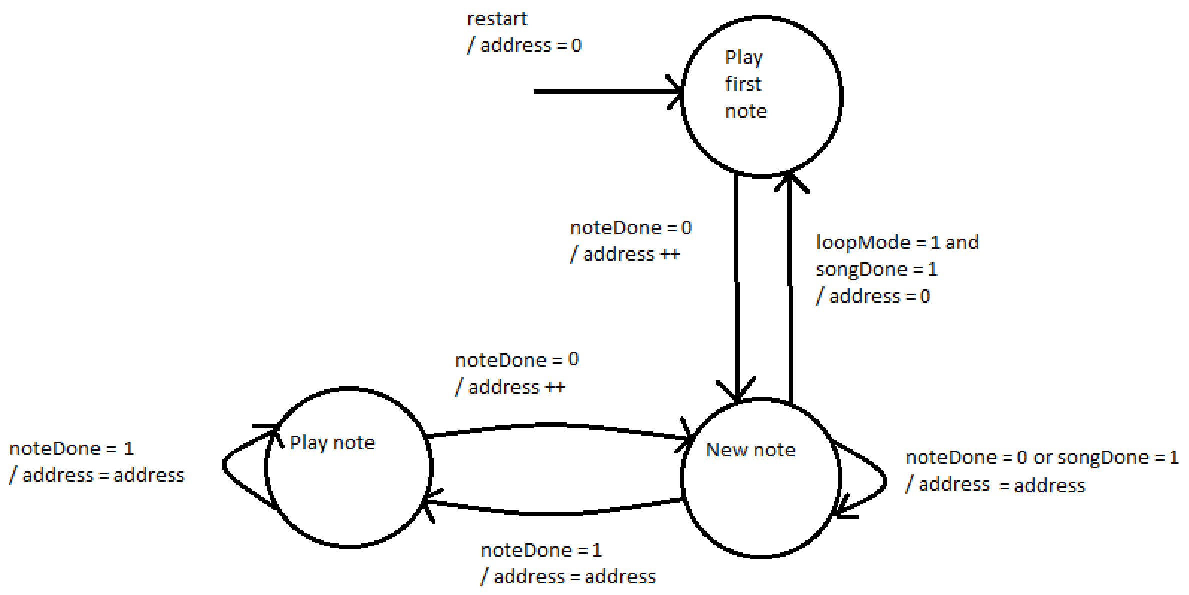


Figure 12 - Complete FSM with end-of-song handling

We simulated this FSM using a simple periodic test input instead of the GATE signal generated by the rest of the circuit. We saw that the circuit indeed entered the loading state and the waiting state properly (see bottom row of simulation), and that as a result the counter was incrementing properly. We checked each of the notes against their values in the ROM and confirmed that the right notes were being loaded using the address.

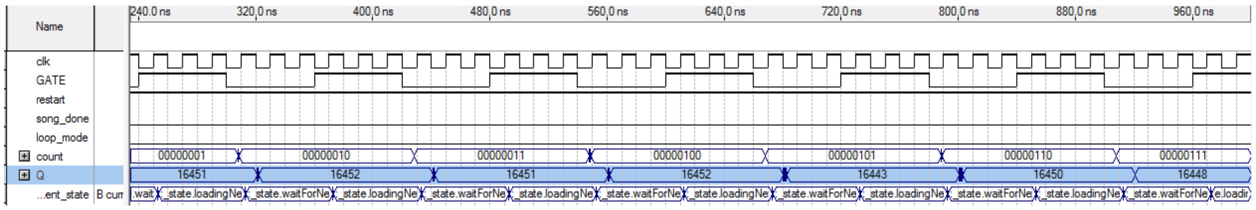


Figure 13 - FSM loading notes from Fuer Elise and waiting in response to test GATE

Pause

We obtained pause functionality accidentally when we were debugging our circuit. Despite our FSM performing well under simulation by itself, the circuit wouldn't play when we connected all the components in the datapath to the controller. We suspected that the problem lay in the reset input to some of the components (having experienced this in previous labs where an active-high pushbutton held a reset value high when we wanted it low). We added a NOT gate to the reset input, but that didn't fix the problem. We checked all of the components and found that the two filtering components (audio_interface and decimator) had expected reset high. When we fixed them, we found that the circuit would only play when we held the reset button, so we removed the NOT gate and realized that we had made a pause button.

The reason "reset" works as "pause" is simple: every component except for the controller itself has a reset input. That means that when reset is asserted, every component stops functioning except for the controller, which stays in whatever state it's in without changing the value stored in the counter. That means that when the rest of the components get un-reset and GATE starts cycling again, the controller takes up right where it left off and plays the next note.

Testing

The first problem we encountered was the counter in our finite state machine. We noticed testing the FSM by itself that the counter would increment several times during one GATE trough (at this point, we were using GATE = 1 rather than GATE = 0 as the transition with increment, hence the waveforms below).

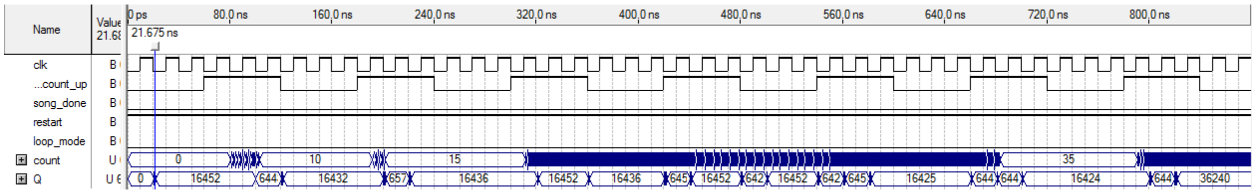


Figure 14 - Counter incrementing multiple times

We checked that the circuit was behaving this way for different initial conditions and lengths of GATE:

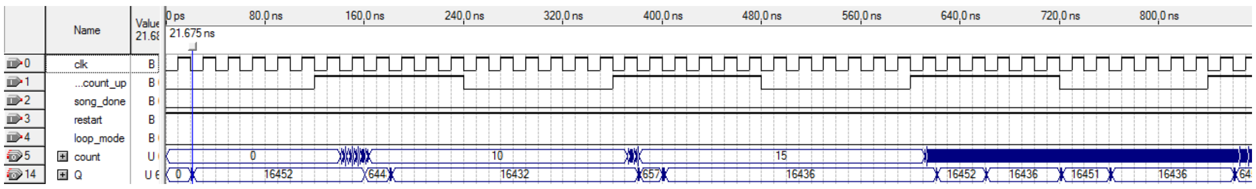


Figure 15 - GATE starts at 0

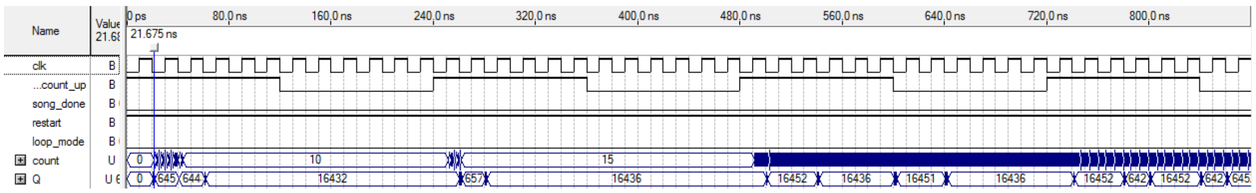


Figure 16 - GATE starts at 1

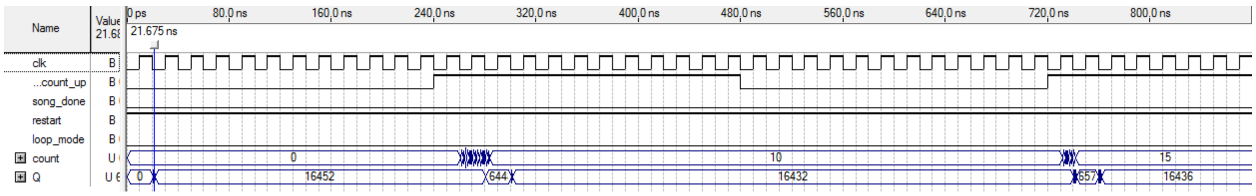


Figure 17 - GATE elongated, starts at 0

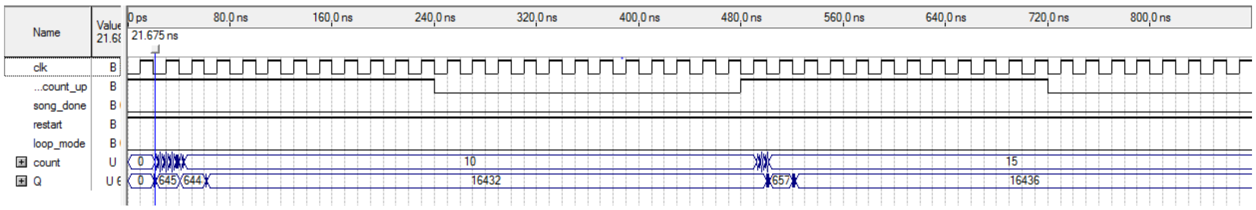


Figure 18 - GATE elongated, starts at 1

Curiously, the incrementing always stopped 10, 15, 35, and 50. We did not have enough time to diagnose this, but we suspect that integers in a process block might be synthesized in hardware in such a way that glitches are likely or that otherwise innocuous delays in our FSM are causing the glitch. We replaced the integer counter with an enable signal to an LPM counter. After that, the FSM worked perfectly fine.

We tested all the other components in the datapath in other labs, which made us think that they would all work perfectly well when put together, but our note timer circuit had a flaw in it which we had not noticed and which caused us a lot of trouble. For some reason, transitions between notes with note duration 000 and note duration 001 result in the first note being dramatically shortened. We figured out that the note timer circuit was the problem and began to debug it.

The note timer contains a process block which sets GATE high for half of the number of tempo enable pulses dictated by note_duration. We used simple integer division to check for that halfway point, that is:

if tempo_enable = '1' and count > (to_integer(unsigned(number_of_pulses))/ 2)

For large values of count, this approach works, as is evident in simulation waveforms:

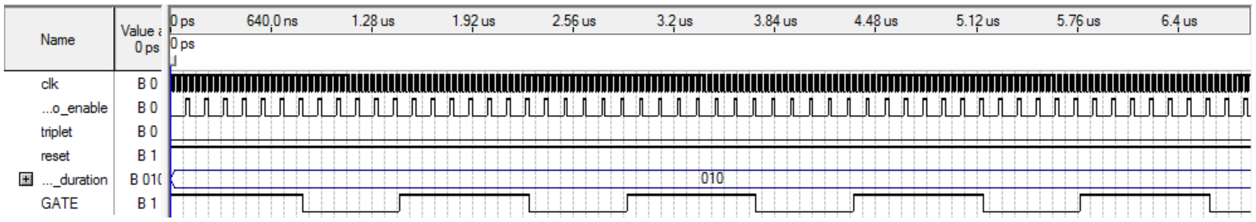


Figure 19 - Note Timer simulation, note duration = 010

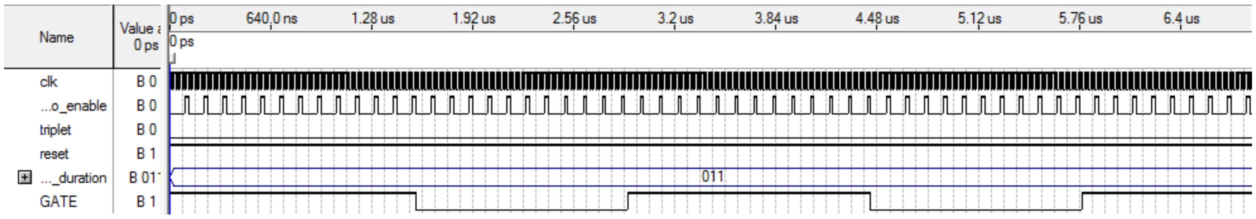


Figure 20 - Note Timer simulation, note duration = 011

Smaller values of note_duration, though, switch GATE values too soon:

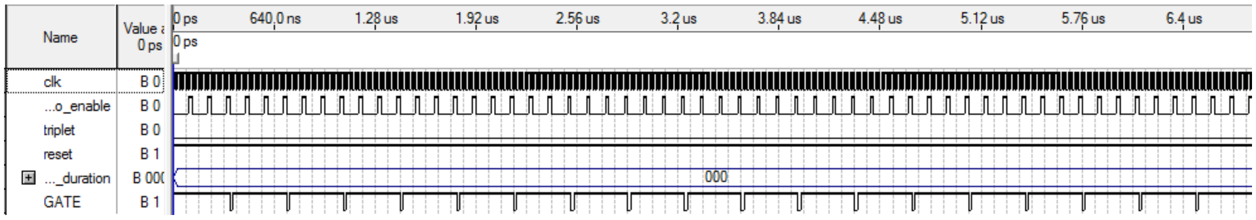


Figure 21 - Faulty note timing with note duration = 000

This happens because (to_integer(unsigned(3))/ 2) is itself not an integer and gets rounded up, so count (decremented) > the quotient for one or two clock cycles too many. This means that rather than having a full GATE cycle from falling edge to falling edge, a note of duration 000 only gets a tiny amount of time to play.

We work around this by checking if the number of pulses = 3 or 2 and handling those cases with simple equality checks, i.e. if count > 2 GATE = 1 for number of pulses = 3 and if count = 1 GATE = 1 for number of pulses = 2.

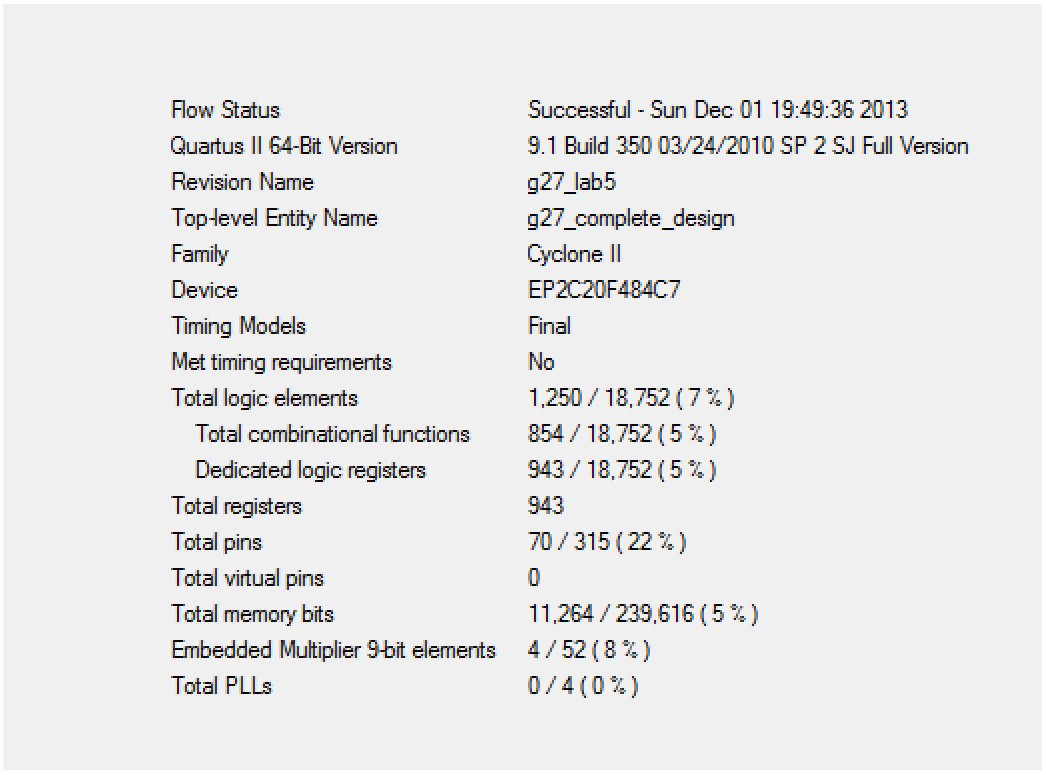
Design of User Interface

We had several design choices to make for the configuration of inputs and outputs on the board: what controls users would have over the song, and how to display the system status (paused, looping, beat indicator). We decided to use the seven-segment LEDs to display 'L' for loop mode and 'P' for paused because these letters can be easily interpreted by someone playing with the circuit, and seven LEDs lighting up to form a letter are much more visible and understandable than a single LED (or other output). We picked the green LED for the beat to distinguish it from these red LEDs.

Ironically enough, even though the subcircuits for this part of the music box are some of the simplest (a multiplexer selects either ‘0’ for the appropriate segments of L if in loop mode and P in paused, lighting those segments up, or all ‘1’, leaving them dark; the tempo circuit controls the green LED as well as the note timer), this is the second most important part of the circuit (after the song output). People using the circuit will think of the status display and the song as the entirety of the music box, just as some people who don’t know much about computers will refer to the monitor as a computer, since it is the most engaging and informative part of the device. The FPGA is the beating heart of our design, and within it the music box controller, but most people (including the authors of this report before they took ECSE 323) couldn’t identify the FPGA on the Altera board, much less say what an FPGA can do. This is why the user interface is so important for the circuit: the flashing lights, for most users, ARE the circuit.

FPGA

The usage summary is as follows:



```
Flow Status                        Successful - Sun Dec 01 19:49:36 2013
Quartus II 64-Bit Version          9.1 Build 350 03/24/2010 SP 2 SJ Full Version
Revision Name                      g27_lab5
Top-level Entity Name              g27_complete_design
Family                             Cyclone II
Device                             EP2C20F484C7
Timing Models                      Final
Met timing requirements            No
Total logic elements               1,250 / 18,752 ( 7 % )
  Total combinational functions    854 / 18,752 ( 5 % )
  Dedicated logic registers        943 / 18,752 ( 5 % )
Total registers                    943
Total pins                         70 / 315 ( 22 % )
Total virtual pins                 0
Total memory bits                  11,264 / 239,616 ( 5 % )
Embedded Multiplier 9-bit elements 4 / 52 ( 8 % )
Total PLLs                         0 / 4 ( 0 % )
```

Figure 22 - FPGA utilization

Part 3: Discussion and Improvements

The mistakes we made making our circuit cost us a lot of time: rather than testing the finite state machine before connecting it with the other components and putting it on the board, we programmed the board, thinking that we wouldn't have any problems since all of our components worked well, and then had to work our way down through the circuit debugging until we figured out that the FSM was the main problem.

Some of the problems in our circuit we could not have anticipated (such as the integer counter problem, which we still don't understand), but Mehdi recognized immediately that the integer might give our circuit problems. We would have gotten through the lab much more easily if those problems had been recognized more quickly.

The musicbox, we realized, is a complete computer: it has a datapath (hardware sections 2-5), a controller (hardware section 1), instructions (song ROM), the ability to branch conditionally to one of those instructions (restart, song selection, loop mode, pause), processable inputs from a user (board switches), and outputs based on the state of the machine (light, sound). It was an exciting realization.

In the final sections, we discuss some of the improvements we would be interested in making to our circuit: dotted note lengths, and chords.

Dotted Note Lengths

In musical notation, a note with a dot beside it has three halves the length it would have if it were missing the dot. For instance, a dotted quarter note has three halves the length of a quarter note, or a quarter note and an eighth note. Currently, the music box can't play dotted-length notes. One can effectively add them by adding a triplet marker to each of the non-dotted notes in a song and leaving the dotted notes without the triplet marker (this is how we made our demo song, "Careless Whisper" by George Michael), but this requires a tempo change, violates the modular principle of good design, and doesn't allow for actual triplets in the rest of the song.

Another possible workaround is playing a note of half the value of the note to be dotted after that note, but this has problems too: the enveloping effect would cause a slight break in the middle of the note, and turning off enveloping for the course of that note would mean that that note alone among all the other notes would not have the proper enveloping.

The workarounds are inadequate. The current song ROM and note timer design could be altered to include a "dotted" bit: if this bit were marked for a note, the music box controller could send a signal to the note timer to multiply number_of_pulses by 1.5 (or simply select a different set of numbers already multiplied by 1.5).

Chords

The circuit can only play one note at a time. There is no workaround for this. But adding chords is very easy. Each song needs N ROMs (where N is the maximum number of notes played at the same time during the course of a song, usually three or four) with each ROM synchronized to the other ROMs for that song. The circuit needs an extra copy of the section 2, 3, and 4 hardware (see above, with the exception of the DSP components). The sqrwave outputs should be scaled (by a factor proportional to N) and then added to each other, and that summed signal would be the input to the audio interface. The scaling by N would ensure that two notes playing at once would not overwhelm the speakers.

System Display

There are a few other things which could be displayed on the 7-segment displays; for instance, a 1 or 2 could indicate which song. The LEDs above the slide switches are unused and could be used as a binary visual counter indicating which note of the song was being played at that moment. This would require a few changes to our controller module like an std_logic_vector output of the address.