Polyphonic Digital Synthesizer

Grant Larson and Catherine Slaughter

Abstract

For our project, we created a polyphonic digital synthesizer with a MIDI interface for MIDI keyboard input. The synthesizer receives 3-byte packages of MIDI data from a MIDI keyboard using the UART serial protocol. For each of the 25 keys on the keyboard, the FPGA stores whether the note is on or off. For each note that is on, a sinusoidal waveform is generated. The resultant waveforms are added and output to a speaker. The synthesizer can play up to 25 notes simultaneously, pitched from C3 through C5.

Table of Contents

1. Introduction

Our project is to build a polyphonic digital synthesizer with a MIDI interface. The synth takes MIDI keyboard input and outputs the sine wave corresponding to each note. If multiple notes are playing, the synthesizer adds the appropriate sine waves.

2. Design Solution

2.1. Specifications

Our circuit converts MIDI keyboard input to audio output (Appendix A; Appendix B, Figure 1). The MIDI keyboard is connected to the Basys 3 FPGA via a 5-pin MIDI connector. MIDI data is received as UART serial at a baud rate of 31250, the MIDI standard. The audio is output from the FPGA to Pmod DA2 as 16-bit SPI-like serial. The analog output of the DA converter is transmitted to a speaker, which outputs the synthesized audio. The system is clocked at 1 MHz. Audio is sampled at 50 kHz.

2.2. Operating Instructions

Operation of our polyphonic MIDI synth is simple. Once the keyboard is plugged in and turned on and the FPGA is powered up and programmed, it is good to go. Pressing the keys will cause the corresponding notes to be played as expected, with the most recently changed (on or off) note number showing up in hexadecimal format on the FPGA 7-segment display. There is no need to flip any switches on the FPGA. It is important that the user does not change any of the default settings on the MIDI keyboard (such as the pitch wheel or octave setting) while the synth is in use. Such

features do not send MIDI codes in the 3-byte packages that our MIDI interface expects. Doing so will throw the controller out of sync and the synth will need to be reset.

2.3. Theory of Operation

We designed our polyphonic synth to be as modular as possible. Doing so made it easier to test each part of our design as we went along, and also made building and adding new features much simpler. Our final design has four independent modules under the same shell (Appendix B, Figure 2). The first is the UART interface, which deals with taking in and synchronizing the serial MIDI data coming from the keyboard, packaging it up into 8-bit bytes, and parallel loading it into our MIDI interface. The MIDI interface takes in the 3 bytes, interprets each as necessary (i.e. reading a note on/note off signal from the message byte), and passes the important info from each on to our sound engine. In the case of our synth, only the first two bytes (the message signal and note number) are actually passed on. However, our design is such that the third byte (the velocity) is also registered, so it would be reasonably simple to add on new modules and features that utilize the velocity data. The next module in our design is our sound engine, which deals with interpreting the note on/off and note number signals from the MIDI interface, keeping track of which notes are pressed, and generating and adding sine waves. It passes this data to our final module, the DA interface, with turns our parallel sine wave data into serial data to be passed along a SPI interface that the digital to analog (DA) converter Pmod can understand. A more in-depth description of each module can be found below.

UART INTERFACE (Appendix B, Figure 3)

The UART interface loads serial data in from the keyboard to be output along an 8-bit bus to the MIDI interface. We use a typical interface here, not entirely unlike that used in the SPI bus lab. First, the serial data gets pushed into a shift register. Then, the shift register parallel-loads its data into an output register, at which point the midi_done_tick is asserted for the MIDI interface. The difficult part of UART is that unlike the SPI bus interface we used in lab, UART does not pass along a clock, so we had to synchronize. To do this, we used a double-flop synchronizer and a clock divider. The clock was divided down based on the agreed-upon baud rate of the MIDI UART data, which is 31250 bits/second. When the serial input goes low, the UART controller (Appendix C, Figure 1) enters a calibration state and identifies the center of the start bit using the middle count of the clock divider counter. The controller then moves to the shift state and continues shifting at the center of each bit, ensuring that a small timing error would not impact the values read in by the UART interface. When 10 shifts are complete (1 start bit, 8 data bits, 1 stop bit), the serial data line returns high, and the shift register is parallel loaded into the output register.

MIDI INTERFACE (Appendix B, Figure 3)

The MIDI interface works by taking in 3 bytes of MIDI data from the UART interface one at a time, and multiplexing them into the appropriate registers for the message, note number, or velocity signal. A counter (which goes from 0-2) outputs the

current count as the select bit to the multiplexer and informs the controller of which byte is being loaded next. When the controller receives an rx_done_tick signal from the UART interface, it checks what the current count is. If the count is 0, the message register loads. If the count is 1, the note number register loads. If the count is 2, the velocity register loads. Each time a register is enabled, the count is incremented by 1. The count rolls back to 0 after the velocity byte has been loaded. Once all three registers have been updated with new data, the MIDI controller (Appendix C, Figure 2) outputs a midi_done_tick for the sound engine. For our synth in particular, the interface runs the message signal through a lookup table to decode the note on/off message[1] to a one-bit signal -- 1 for on, 0 for off. It outputs this new signal and the 8-bit note number to the sound engine.

SOUND ENGINE (Appendix B, Figure 4)

The sound generator module takes in the note on/off signal and note number from the MIDI interface, along with a take_sample tick that is generated in the shell. This signal is generated by dividing the 1MHz clock down to a 50kHz signal. It is crucial to the sound engine's timing that the take_sample tick has a frequency smaller than that of the master clock by several orders of magnitude.

The sound engine begins with a note on register that functions similarly to a map or a dictionary. When the engine receives a midi_done_tick signal from the MIDI interface, it stores the value of the note_on signal at the location along the register

[1] By MIDI convention, the first 4 bits of a note on message are 1001 (9 in binary), and the first 4 bits of a note off message are 1000 (8 in binary)

corresponding to the note_num signal coming from the MIDI interface. The note_num is the key and the note_on signal is the value. As the data comes in from the MIDI interface, a count of the number of keys currently being pressed is kept, increasing with each note on and decreasing with each note off. This is used later to decrease the volume of the final output. The note numbers sent by the keyboard range from 48 (C3) to 72 (C5). We subtract the value of the lowest note number, 48, from the note number coming in, so our note on register is indexed from 0 to 24.

Each bit on the register acts as the enable bit for one of 25 phase accumulators, one for each key of the keyboard. The phase accumulators output a value between 0 and 4095 (a 12-bit number), where each possible output corresponds to a value in the sine lookup table. In our code, we were able to build the phase accumulators using a VHDL generate statement, instead of hard coding each one individually. The specifics of each phase accumulator can be found in the **Phase Accumulators** section below.

In order to achieve polyphony, the sine waves for each of the pressed keys must be added together. Each key corresponds to a sine wave of a given frequency, and the sound of several keys is the superposition of all the corresponding waves. To implement polyphony, we take advantage of the several-order-of-magnitude difference between the frequencies of our system clock and our take_sample signal. Because the individual note phase accumulators only update on the take_sample signal, we can cycle through and sum up the values in a sine accumulator in between take_sample ticks using the system clock.

The sound generator controller is in charge of running this whole process (Appendix C, Figure 3). The controller contains a counter that runs from 0 to 24, incrementing on the 1MHz clock. Each time a take_sample tick is received, the counter begins incrementing, and stops after terminal count is reached. The count of the counter acts as the select bit (called count_mux on our block diagram) for a multiplexer, which takes in the current values of each phase accumulator and outputs them to the sine wave LUT (Appendix I). From here, the output of the sine wave LUT is input into a sine accumulator. After terminal count is reached, the controller goes into a load state. The total value stored in the sine accumulator is divided by the number of keys that are currently being pressed -- such that the volume does not increase as the number of keys pressed does -- before being stored in a register to be output to the DA interface. The sine accumulator sums 25 12-bit numbers from the phase accumulators, so it outputs an 18-bit number. However, dividing by the number of keys pressed ensures that the output always ends up being 12 bits, matching the width of the DA converter. All of these operations (multiplexing, sine lookup, accumulating, and loading into the register) are run on the 1MHz clock and occur before the next take_sample tick comes, so they do not impact the way the final output sounds. The DA interface is also run on the slow take_sample clock. The final output register is absolutely crucial because the sine accumulator must be reset to 0 before the next take_sample tick, but the data also needs to be available at the same take_sample tick.

While the logic for adding up the sine values of all 25 phase accumulators is not too difficult, it is a bit tricky to clock. This is because there is an inherent latency present

in the sine wave LUT. In our case, the latency is 6 clock cycles. In order to deal with this, we have to delay any outputs from the controller that go to parts of the data path after the LUT, specifically our accum_en and load_en signals. In order to do this, we fed the outputs from the controller through shift registers of length 6. For each signal, we pushed the value from the controller into the MSB of each register and popped the LSB as our accum_en_delay and load_en_delay, respectively. By making the length of the registers equal to the latency of the LUT, we ensured that the signals reached their respective places six clock cycles after the controller updated them. Doing so ensured that we did not begin to accumulate data too early (thereby accidentally adding old data), nor did we stop accumulating too late or load too early (missing the last 6 sine values). This made our controller design simpler, as we didn't need to include weird waiting states where counting had finished but we weren't ready to load the data.

Phase Accumulator

The individual phase accumulators in our sound engine take in the value of note_num_reg at the index for their corresponding key on the board and the take_sample signal from the shell. Each accumulator outputs a value from 0 to 4095 to be put through the sine LUT. For each accumulator, although it is being clocked on the 1Mhz clock, it will only update when both its corresponding note_on signal is high and when the 50 KHz take_sample signal is up (which it only is for a single 1MHz clock cycle). The step value the accumulator increases by (called M) is passed in as a generic from the sound generator at the generate statement and does not change. The M values

are hard-coded into a constant array of integers in our sound generator program. They were calculated according to the Direct Digital Synthesis method for generating sine waves, as outlined on the Canvas page "How to make a sine wave (revised)" written by Professor Hansen. For our synthesizer, our sampling frequency is 50kHz, our N is 4096, and our F_0 (the frequency of the first note on the keyboard) is 130.81Hz. The calculations of our 25 M values were done in MatLab (Appendix K).

We found that our chosen values of N and F_s gave us one or two places where the M values calculated for two neighboring notes was the same, because M has to be an integer. To retain precision and eliminate this issue, we multiplied all our N values by 1024 (a 10-bit shift) and made our phase accumulators 22 bits wide. The accumulated value is then shifted back 10 bits before being output to the sound generator multiplexer.

DA INTERFACE (Appendix B, Figure 5)

The DA interface turns our 12-bit bus of data from the sine wave accumulator and turns it into a SPI-like data stream, which that DA Pmod converts to analog data for the speaker. It loads in the data on the take_sample tick, before shifting the bits out one at a time on the 1MHz clock, well before the next take_sample tick comes in. The low-true spi_cs signal is handled by the DA controller (Appendix C, Figure 4), and the 1MHz spi_clk is passed on to the DA converter by the shell.

2.4. Construction and Debugging

After spending a lot of time designing three of our four modules, we first built our project in three overarching pieces, our input modules, output modules, and sound generator. We first built our input modules, the UART and MIDI interfaces, wired them together and simulated each section to verify that data was being read, interpreted, and passed on correctly (Appendix D, Figures 1-4; Appendix G). We did not do a physical test until later in the process. Next, we wrote the output modules, beginning with the DA interface. To test the DA interface, we also wrote the PA module that we were planning on using in our sound engine, and wired the two together in a shell such that the speaker would play middle C. We then conducted a physical test of this section, using the oscilloscope to make sure data shifted out correctly (Appendix D, Figures 5-7). Once every other piece of our synth was verified, we started building the sound engine. We were exceptionally meticulous in writing the module code and while wiring all our modules up in our top-level shell, so much so that we did not end up needing to testbench that piece of the project.

Thankfully, we did not have to spend too much time debugging any one part of our project. Obviously, there were many small typos and such that were easy fixes, so we will outline only the most significant and/or frustrating bugs here. For the input modules, our physical test revealed a bug that had not surfaced in our simulation. We did not perform a physical test of this piece until after we had built and tested our output modules, and during this time we decided to change our master clock from

10MHz to 1MHz. This caused an error in the clock divider for the UART interface. This bug was easily fixed once discovered. We spent more time debugging the output modules. Ultimately, we found two primary bugs. The first being that we had failed to instantiate the phase accumulator with a starting value of 0, so when it went to accumulate it was adding M to undefined. The second was an error in our XDC file where we tried to map the digital audio data to two different ports, so the DA converter could not receive it properly. When we went to run our final project, the first two things we checked were the sine accumulator's initial value and the XDC file. We had once again made errors in each, but these were the only remaining bugs present. Had we encountered more issues, we would have simulated our sound engine module, but through meticulous coding we were able to avoid doing so.

3. Justification and Evaluation

We found our design to be generally efficient, with minimal resource utilization and robust polyphonic functionality. Through use of a generate statement in the sound engine module, we were able to efficiently generate the 25 accumulators necessary for the full range of a 25 key MIDI keyboard. The less efficient alternative would be to instantiate each accumulator separately. We made some trade-offs between efficiency and sound quality in our phase accumulators, where we added in the 10-bit shift on the M values then divided back down. Obviously this division takes some computational power, but we decided it was worth it so that each note had a unique frequency.

The biggest improvement that could be made upon our design is dividing the added sine waves by the square root of the number of keys pressed rather than number of keys pressed, so the volume is not decreased by pressing multiple keys at once. As is mentioned in the comments of the sound engine code in Appendix G, we attempted to handle the square root by creating a coarse LUT. However, we found that the dividing by less than N caused the sine sum to overflow, resulting in noise. At that point, we decided in the interest of time that the square root functionality was best left for a future iteration.

4. Conclusions

In the end, our final project went above and beyond the expectations we set in our original design. Our initial design was a simple monophonic synthesizer without any bells or whistles, designed to be intentionally modular so we could potentially try for polyphony later in the process. We spent the majority of our time on this project trying to design and implement the datapath and controller for our polyphonic sound engine module, so finally getting it to work was very satisfying. Although we achieved more than we had hoped to for the purposes of this project, it is important to note that our design is intentionally modular so new functionality could easily be swapped in. Given all the time in the world, we could potentially add in features such as an interpretation of the velocity byte to augment volume, string sounds, or a low-pass filter to clean up the sound output. These are just a few of a seemingly endless number of additional features that could be added to our design.

Given the chance, our strongest recommendations to future groups considering a polyphonic synth would be to make your design as modular as possible, and to start early designing the logic for your sound engine. Don't be afraid to ask lots of questions. In general, we'd recommend groups take the time to sit down and work through the data path(s), controller(s), and *clocking* of your system very carefully before you go about writing any code. Additionally, we recommend you code systematically, with lots of comments. Most of our modules were carefully coded moving from left-to-right on our block diagrams. Both of these practices slashed the time and frustration we had to spend on debugging and writing our report.

5. Acknowledgements

Thank you to our project TA Evan for his time, willingness to answer questions, and general support from the very first steps of our project. We would also like to thank Dave and Dave for their patience and expertise in the lab. Finally, we would like to thank Professor Hansen for his guidance, insight, and enthusiasm throughout the duration of ENGS 31 and the project process.

Throughout the project we generally worked on things together, as opposed to dividing up the work. All of our data paths and state diagrams were designed by both of us with much, sometimes heated, discussion. When in the lab, we would take turns typing the code and guiding the typer, so as not to get too distracted on any one section. Grant

wrote the UART and DA interfaces, which we both worked to debug. Working and coding together in this way was important for the overall flow of our project, because it retained consistency in how the different modules interact with each other. We both contributed to writing this report. Grant made the data path diagrams, and Catherine made the state machine diagrams. In general, Grant pulled most of the appendix together and Catherine focused on writing most of the report itself, with plenty of collaboration and overlap.

6. References

Our most important references we used in this project were the Canvas page "How to make a sine wave (revised)" written by Professor Hansen, and the old UART interface lab that was used in previous years of ENGS 31. Any resources from the internet that we used can be found in the bibliography below.

J., Byron. "MIDI Tutorial." *Sparkfun*, 8 Oct. 2015, learn.sparkfun.com/tutorials/midi-tutorial/all.

"MIDI Note Numbers and Center Frequencies." *Inspired Acoustics*, www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies.

Murphy, Eva, and Colm Slattery. "Ask The Application Engineer-33: All About Direct Digital Synthesis." *Analog Dialouge*, Aug. 2004, www.analog.com/en/analog-dialogue/articles/all-about-direct-digital-synthesis.html.

"Musical Notes." *Musical Note Frequencies*, TechLib.com, www.techlib.com/reference/musical_note_frequencies.htm.