Jukebox Music Synthesizer

Microprocessors Project Final Report

George Kuan and Adrian Mettler

December 13, 2003

Contents

1 Introduction

The Jukebox Music Synthesizer is a two-channel digital music synthesizer that can play a variety of songs selectable by the user. The system is comprised of a programmable sound generator (PSG) that resides on the Xilinx FPGA and a song database on the PIC microcontroller. The PIC also includes control code to select which song to play, read it from memory, and send it to the FPGA for synthesis. The synthesizer is capable of generating square waves and approximations to sawtooth, triangle, and sine waves as output. Presently, the song database hosts four songs: "White Christmas", "I've been working on the Railroad," the Overworld Theme from the Super Mario Bros. video game, and "Für Elise." The set of songs is easily extensible and is practically only limited by the PIC's program memory.

2 Song Encoding

The MIDI note system consists of 127 distinct notes ranging from very low (8 Hz) to very high (12,000 Hz). Our song encoding allows specification of all MIDI note values with the exception of zero which is used to indicate a rest.[1] The present system stores a full lookup table of these note values in the PIC's EEPROM. Each note corresponds to two consecutive bytes in the EEPROM. This is a 16-bit value that corresponds to the period in terms of clock cycles (0.5 μs for 2 MHz). This is used by the FPGA to determine how fast to synthesize the waveform for the note. We computed the wave period using the formula $\frac{2,000,000\text{Hz}}{\text{Frequency of MIDI Note}}$.

The table of notes, frequencies, and periods is included as an appendix. Note that MIDI numbers below 23 have periods longer than 65,535 cycles, and thus cannot be represented in 16 bits. In the EEPROM, these delays are stored as 65,535. This is acceptable since none of our songs use notes this low (more than three octaves below middle C). Such notes would likely be inaudible with our amplifier and speaker anyway.

Our song encoding scheme allows 16 possible synthesis modes to be specified for each

[1] MIDI timestamps individual notes and only specifies ones that are actually present, thus has no need for a rest value.

note on each channel, using a 4-bit code. Four codes have been defined are are understood by our synthesis logic on the FPGA. These are 0 for square wave, 1 for approximate triangle wave, 2 for approximate sawtooth wave, and 3 for approximate sine wave.

We encode notes in the following manner:

8 bits	8	8	4	4
Duration	MIDI Note 1	MIDI Note 2	Synth Type 1	Synth Type 2

1. The first byte of a note is the duration of the note. The units of the duration is dependent on the clock speed at which the FPGA is running. At 2 MHz, the duration is in terms of 1 unit $= 16.4ms$.

2. The second byte is the MIDI number of the note for the first channel.

3. The third byte is the MIDI number for the second channel.

4. The last byte indicates the waveform synthesis parameters for the two channels.

Rests are encoded using 0 in place of the MIDI number.

The songs are stored in the program memory of the PIC. We chose to use the program memory for storing songs because it is the largest memory available to us. Songs consist of the notes in the song sequentially encoded in the format specified above. The end of the song is indicated by a 0 in place of the duration, followed by the address of the beginning of the song in program memory. This format facilitates easy looping of songs. When a new song is selected by the user, it is necessary to find it in program memory. For this purpose, we have a song list at a constant address (0x200). Each entry is at an offset equal to the song number left shifted by two, and is in the same format as the entry at the end of a song, so the same code can be used to interpret it.

3 PIC Song Selection and Note Interpreter

The portion of our project implemented on the PIC microcontroller utilizes interrupts heavily. There are two main interrupt handlers for the song selection and note-period translation/transmission system.

1. Timer 0 is used to measure discrete timesteps of 16.4 ms. The number of timesteps remaining in the current note is stored at a memory address and decremented each time this interrupt occurs. If this count reaches zero, the next note is read out of the table. Each MIDI number is looked up in the EEPROM address calculated by shifting the note number left one, and the 16-bit period values are read out and transmitted to the FPGA. The synthesis parameter byte is then transmitted to the FPGA verbatim. Transmission to the FPGA uses the onboard SPI functionality, with the PIC in master mode. Since five bytes must be transmitted closely one after the other at one bit per instruction clock, nops are needed to delay the sending of bytes following the first.

   If the record read indicates that the end of the song is reached instead of a note, the table pointer is set to the specified address (indicating the start of the song) and the note interpretation code is re-run at that address. This is how song repeating and the song list work. Once this interrupt handler is complete, it triggers A/D input.

2. The A-D converter interrupt: The A/D converter is used for user input to the jukebox. When triggered, the new song number is determined by the high-order bits of the A/D result. The system currently requires the two most significant bits because we have four songs in the database. If songs are added, the total number must be a power of two and the appropriate number of bits must be used for song selection. The song number will be compared with the number of the song currently playing as indicated by a variable in memory. If it is the same, nothing happens (the song continues playing), but if it differs, the current song variable is updated, the table pointer is set to the appropriate entry in the song list, and the time remaining on the current note is set to 1. This way, the next time the clock interrupt triggers, playback will immediately start on the new song.

Upon starting up, the PIC program configures the timer, table pointer, A-D converter, and respective interrupt handlers. It also selects a default song and begins to play it (it will be interrupted almost immediately if the knob is set to play a different song, however). It then enters a null-effect loop waiting for the above interrupts to be triggered.

4 Waveform Generation in the Sound Generator

Generation of notes is performed independently for the two synthesis channels. Each one has a register storing the note period in clock cycles and the synthesis type for that channel. Synthesis for each channel is identical, and uses the following procedure.

4.1 Slow Counter

Generation of all waveforms is based on a counter that increments each time approximately 1/16 of the specified note period has elapsed. This is implemented as a collection of four independent counters that increment on each clock cycle.

- The main counter resets the output and all the counters to zero when its value reaches the note period. It also increments the count and resets the other three counters to zero when its value is equal to the note shifted left by 1.
- The second counter increments the output and resets the third and fourth counters when its value reaches the period shifted left by 2.
- The third counter increments the output and resets the fourth counter when its value equals the period shifted left by 3.
- The last counter increments the output when its value reaches the period shifted left by 4.

All counters are large enough to only trigger once before being reset by some other counter. The end result is that the output is a discretized indicator of the current phase of the waveform. The multiple-counter method of synthesis gives us the most accurate values possible without requiring division or addition logic, which would prevent two channels from fitting on the FPGA.

4.2 Square Waves

Square wave synthesis is performed by simply setting all four bits of the waveform output to equal the highest order bit of the slow counter. This gives a signal that is zero half of the

time and has the maximum value the other half of the time, which is precisely the square wave we wish to synthesize.

4.3 Sawtooth Waves

Sawtooth waves simply use the unmodified values of the slow counter. The phase value it gives is used directly as a sawtooth wave of the correct frequency.

4.4 Triangle Waves

We used the following approximation to a triangle wave:

Slow Counter	Triangle	Visual
0	0	
1	2	==
2	4	====
3	6	======
4	8	========
5	10	==========
6	12	============
7	14	==============
8	15	===============
9	14	==============
10	12	============
11	10	==========
12	8	========
13	6	======
14	4	====
15	2	==

The values for the output column can also calculated using the formula below, which we used in our Verilog:

$$triangle\_out = \begin{cases} phase << 1 & \text{for } 0 \leq phase \leq 7 \\ 15 & \text{for } phase = 8 \\ -(phase << 1) & \text{for } 9 \leq phase \leq 15 \end{cases}$$

4.5 Sine Waves

Sine waves are generated by direct synthesis, using approximate values for the sine function at each time value of the slow counter. The following set of values is used:

Slow Counter	Sine	Visual
0	0	
1	1	=
2	3	===
3	6	======
4	9	=========
5	12	============
6	14	==============
7	15	===============
8	15	===============
9	14	==============
10	12	============
11	9	=========
12	6	======
13	3	===
14	1	=
15	0	

5 Digital-to-Analog Conversion and Amplification

The PSG's 5-bit output must be converted into an analog signal before the output reaches the speaker. The system utilizes a network of several resistors to accomplish this. The most significant bit of the PSG output, waveout[4], is connected to a resistor with the least resistance R_1. For our design, we used $R_1 = 495\Omega \approx 500\Omega$ because this resistance can be easily obtained by the resistors available in the lab (see schematic of breadboard circuit for details). The second-most-significant bit, waveout[3], is connected to a resistor with approximately twice the resistance $R_2 \approx 2R_1$. We let $R_2 = 1$ kΩ. The third, waveout[2], is connected to a series of resistors with effective resistance approximately twice that of R_2 (i.e., $R_3 \approx 2R_2$) and so on. For this case, we let $R_3 = 2$ kΩ. This configuration of resistors converts each bit into an analog signal of the correct maximum amplitude. That is to say, the outputs corresponding to more significant bits have higher amplitudes than those corresponding to less significant bits. When all these output signals which have suffered the

appropriate amount of voltage drop are shorted together, the result is the analog sum of all the digital outputs weighted according to how significant each bit of digital output was.

If we were to wire this analog sum directly to the speaker, almost all of the power will dissipate in the resistors and hardy any power will be left for the speaker because the speaker has a comparatively low impedance (8 Ω). Consequently, the analog sum must be appropriately amplified. For this purpose, we constructed an inverting operational amplifier circuit in order to amplify our analog sum. We used a 47 kΩ resistor for the feedback to the opamp circuit in order to maximize the gain where gain directly varies with the feedback resistance. Because we needed maximum gain to obtain the maximum volume, we used the resistor with the greatest resistance that we could obtain.

One important concern we encountered was the problem with powering the opamp circuit. The opamp model we were using, the LM741, was not a rail-to-rail amplifier. Consequently, we could not power the circuit with a mere +5 V and expect that the amplified signal to also approach +5 V. The solution to this problem was to connect a +12 V and -12 V power supply to V+ and V- of the opamp respectively. Because the opamp circuit inverted the signal in the process of amplification, we had to connect the other terminal of the speaker to the standard +5 V power supply. The output signal from the opamp is voltage divided against this +5 V with a potentiometer to provide a master volume control.

6 Conclusion

The final Jukebox Music Synthesizer meets our original specifications. We believe that we made good utilization of the PIC and FPGA, using over a kilobyte of program memory and over 75% of the CLBs. Because of this high utilization of the FPGA, we had to limit the temporal resolution of the waveform in order to synthesize two channels. We believe that using only 4 bits for phase may be the reason that synthesized chords do not sound like one would expect. However, we were able to implement a good variety of features and songs in an architecture that could be extended to more songs or synthesis options in the future.

References

[1] Gilbert DeBenedetti. Free Piano Music! Last accessed: December 11, 2003.
⟨ http://www.pitt.edu/~deben/freebies.html ⟩

[2] Koji Kondo. Super Mario Bros. Sheet Music. Last accessed: December 11, 2003.
⟨ http://www.classicgaming.com/tmk/sheet.shtml ⟩

[3] LM741 Operational Amplifier Datasheet. National Semiconductor Corporation: August 2000. Last access: December 11, 2003.
⟨ http://www.national.com/ds/LM/LM741.pdf ⟩