ASEE's 123rd Annual • Conference & Exposition • New Orleans, LA • June 26-29, 2016

Paper ID #15674

Integrating Computer Engineering Labs with a "Sound Theme"

Dr. Pong P. Chu, Cleveland State University

Integrating Computer Engineering Labs with a "Sound Theme"

1. Motivation

Recent engineering education studies call for change to enhance student learning and to better prepare graduates to meet the new challenge[1,2,3]. A good engineer should have a deep understanding of a domain and can apply the knowledge to solve problems[4]. This requires two types of practices – the "component skill," which is the knowledge of a specific domain, and the "integration skill," which applies and integrates component skill to address complex and realistic problems[5]. The Carnegie Foundation for the Advancement of Teaching conducted a five-year study of engineering education and published the results in a book titled *Educating Engineer: Designing for the Future of the Field* [3]. It points out that one deficiency is that engineering curricula mainly focus on the component skill and teach each subject in isolation and without proper context. Students are not adequately prepared for the integration skill. The study recommends a "spiral model" to provide more effective learning experiences:

> "*… the ideal learning trajectory is a spiral, with all components revisited at increasing levels of sophistication and interconnection. Learning in one area supports learning in another*."

The study also calls the labs a missed opportunity and states that[3]:

> *"…[The labs] can be more effectively used in the curriculum to support integration and synthesis of knowledge, development of persistence, skills in formulating and solving problems, and skills of collaboration. Design projects offer opportunities to approximate professional practice, with its concerns for social implications; integrate and synthesize knowledge; and develop skills of persistence, creativity, and teamwork."*

Our work is motivated by the study. Instead of treating the labs as the adjuncts that follow the learning of the theories and presenting them in a limited "component context," we use them as a cohesive framework to connect and integrate the individual courses. The lab framework will keep the lecture content intact but update the experiments and projects to make students aware of the big picture, help them to relate the individual subjects, and apply and integrate the previous learning in a new context.

The labs spread over all hardware related courses, including freshman engineering, introductory digital systems, advanced digital systems, computer organization, embedded systems, hardware-software co-design, and senior capstone design. The complexities and abstraction levels of the experiments and projects gradually grow as students progress through the curriculum. The key concepts are repeated in different courses with increasing sophistication and studied from different aspects and contexts, such as software implementation versus hardware implementation, gate-level design versus system-level integration, etc.

The overall work consists of three "themes." The sound theme is one of the themes and the focus is to use software and hardware to generate a music tone, essentially constructing a *music*

synthesizer[6]. This theme is selected for two reasons. First, most students have a general idea about music instruments and many play some types of music instruments. Thus, they can easily relate to this theme. Second, a music tone is a periodic wave, which is the fundamental function used in circuits and signal analysis, and its generation connects to many subject areas in electrical and computer engineering.

The remaining article is organized as follows: Section 2 provides on an overview of the basic characteristics of a music tone; Sections 3 introduces the concept of music synthesis; Section 4 discusses the hardware schemes used to generate periodic waveforms and the construction of a music synthesizer I/O core; Section 5 illustrates experiments and projects associated with the synthesizer hardware and discusses the methods to integrate them into computer engineering curriculum; and the last section summarizes the work.

2. Overview of a Music Tone

A *music tone* is a steady periodic waveform and is characterized by four aspects:

- *Frequency* (also known as *pitch*).
- *Amplitude* (also known as *loudness*).
- "*Shape*" (also known as *timbre*).
- *Duration*.

A *music note*[8] specifies the frequency and may also contain the duration information. The notes are grouped into *octaves* and their frequencies are doubled after each octave. There are twelve notes in an octave, represented by C, C♯, D, D♯, E, F, F♯, G, G♯, A, A♯, and B. The frequencies from the octave 0 to the octave 8 are summarized in Table 1.

	oct 0	oct 1	oct 2	oct 3	oct 4	oct 5	oct 6	oct 7	oct 8
C	16.4	32.7	65.4	130.8	261.6	523.3	1046.5	2093.0	4186.0
C♯	17.3	34.7	69.3	138.6	277.2	554.4	1108.7	2217.5	4434.9
D	18.4	36.7	73.4	146.8	293.7	587.3	1174.7	2349.3	4698.6
D♯	19.5	38.9	77.8	155.6	311.1	622.3	1244.5	2489.0	4978.0
E	20.6	41.2	82.4	164.8	329.6	659.3	1318.5	2637.0	5274.0
F	21.8	43.7	87.3	174.6	349.2	698.5	1396.9	2793.8	5587.7
F♯	23.1	46.3	92.5	185.0	370.0	740.0	1480.0	2960.0	5919.9
G	24.5	49.0	98.0	196.0	392.0	784.0	1568.0	3136.0	6271.9
G♯	26.0	51.9	103.8	207.7	415.3	830.6	1661.2	3322.4	6644.9
A	27.5	55.0	110.0	220.0	440.0	880.0	1760.0	3520.0	7040.0
A♯	29.1	58.3	116.5	233.1	466.2	932.3	1864.7	3729.3	7458.6
B	30.9	61.7	123.5	246.9	493.9	987.8	1975.5	3951.1	7902.1

Table 1. Frequencies of music notes

There is a simple relationship between two successive note frequencies. Let the frequencies of two notes be f_i and f_{i+1}, then

$$f_{i+1} = 2^{1/12} * f_i$$

The equation implies that a frequency is doubled after one octave (i.e., 12 notes):

$$f_{i+12} = (2^{1/12})^{12} * f_i = 2 f_i$$

For example, the frequency of note C in the octave 1 is twice the frequency of note C in the octave 0. This relationship can help us to calculate the note frequency.

A music note may include information to express the relative duration. The basic unit is a *quarter note*. Its duration is twice the duration of an *eighth note* and four times the duration of a *sixteenth note* and is half the duration of a *half note* and a quarter the duration of a *whole note*. The *tempo* specifies the time interval of a quarter note. It is expressed in terms of BPM (beats per minute). For example, 120 BPM specifies that the interval of a quarter note is 0.5 second (500 milliseconds).

For a tone generated by a real music instrument, its shape is very complex and irregular. This aspect cannot be described by mathematical functions.

3. Overview of Sound Synthesis

3.1 Sound generation

A music tone can be synthesized electronically[9]. The waveform is periodic and continuous and its frequency and amplitude can be adjusted. To emulate the duration aspect, a *gating* mechanism can be added to pass the waveform to the output for a specific amount of time.

The main limitation of the synthesized waveform is in the shape aspect. While the timbre of a real musical instrument is very complex and irregular, a synthesizer can only generate simple and structured patterns, such as a *square wave* or a *sinusoidal wave*. The square wave contains a large number of high-frequency harmonics and its sound is unnatural and unpleasant. The sinusoidal wave is the *pure tone* but its sound is plain and flat. One method to mimic a real music instrument is to modulate the sinusoidal waveform with a *loudness envelope*, which is discussed in the next subsection.

The music synthesis is mainly done by custom hardware. However, software can be used to generate a simple low-frequency square wave. The software approach is discussed in Subsection 3.3 and the hardware synthesizer is discussed in Section 4.

3.2 ADSR envelope modulation

When a note is produced in a real music instrument, the loudness changes over time. It rises quickly from zero and then decays over time. To model the effect, we can multiply the constant tone by a loudness envelope. The *ADSR (attack-decay-sustain-release) envelope* is the most widely used scheme and is the foundation of a music synthesizer[10]. A representative ADSR envelope is shown in Figure 1. The contour of the envelope corresponds to pressing and releasing a key of a music instrument, such as a piano. When a key is pressed, the loudness quickly rises to the maximum (attack segment), then falls (decay segment) to a rather constant level (sustain segment), which is maintained until the key is released. The sound then quickly

fades away (release segment). We can imitate the tones of different instruments and obtain special effects by adjusting the levels and lengths of various segments. After the release segment, the amplitude returns to zero. Thus, the envelope also implicitly performs the gating mechanism. The duration corresponds to the sum of the four segments.

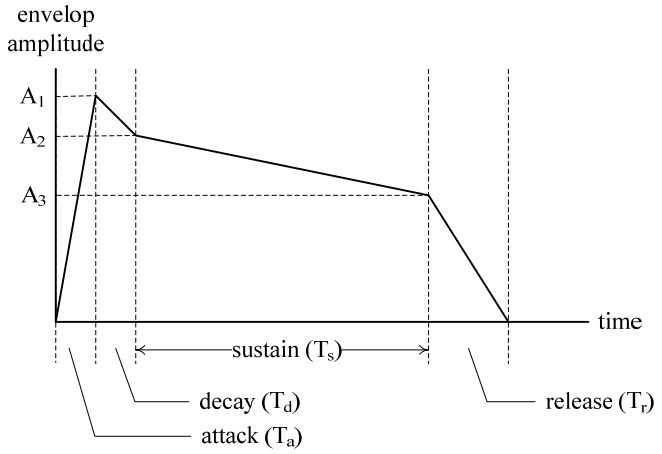


Figure 1. ADSR Amplitude Envelope

3.3 Software Sound Generation

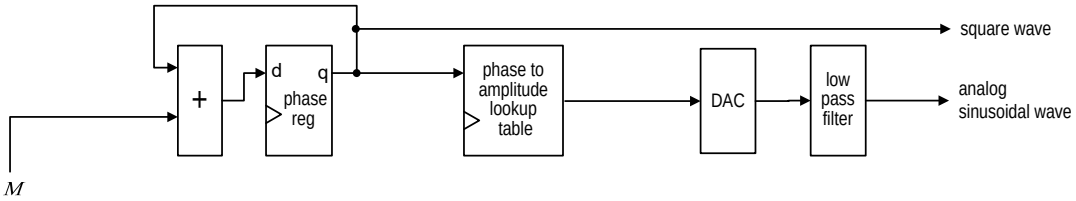
A software program can generate a square wave and control its frequency. A square wave is a one-bit signal that oscillates between 0 and 1 with a fixed interval. For a 10 KHz signal, its period is 100 μs and the signal toggles every 50 μs. A microcontroller or an embedded processor can generate the signal with its general-purpose output port and timer. The program checks the progress of the timer and toggles the output when the timer reaches the designated interval. While the algorithm is simple, it imposes a rigid timing constraint since the signal must be switched in a precise moment. A software program can also control the duration of a note in a similar fashion. However, its interval is around a fraction of a second and thus the timing is not as critical. A program cannot adjust the amplitude or shape of the output wave unless a special sound generation I/O peripheral is included in the embedded system.

4. Hardware Sound Generation

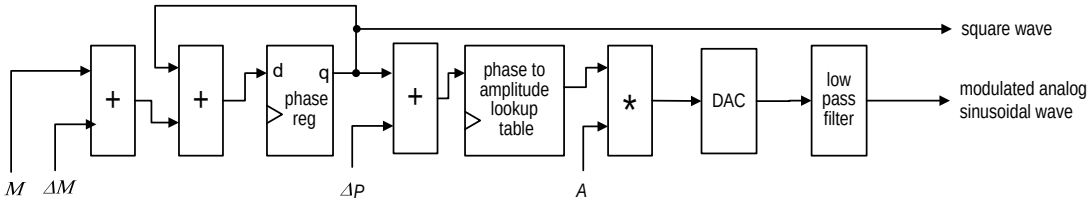
Because of the strict timing constraint, custom hardware is needed to generate the waveform. *DDFS (direct digital frequency synthesis)* is the most widely used scheme for digital implementation[11,12]. It is incorporated into instrumentation as well as music synthesis. The following subsections discuss the development in stages[13]:

- *Digital DDFS* (to generate a square wave).
- *Analog DDFS* (to generate a sinusoidal wave).
- *Modulated analog DDFS* (to generate a modulated sinusoidal wave).
- *Music synthesizer* (to modulate an analog wave with an ADSR envelope generator).
- *Music synthesizer I/O core*.

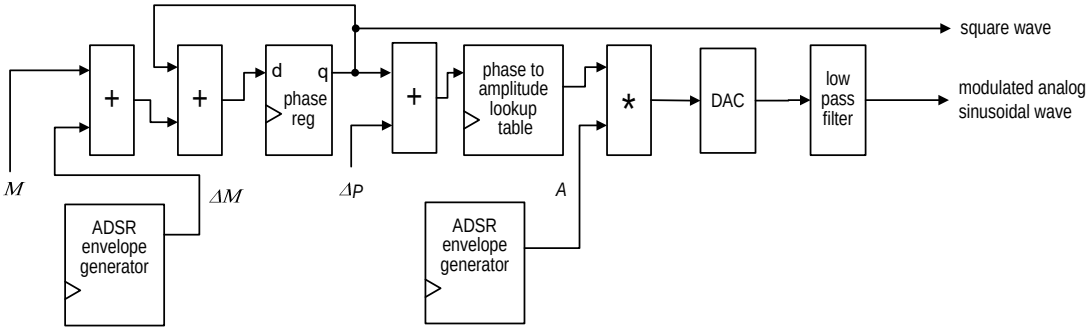
(a). Digital DDFS



(b). Analog DDFS



(c). Modulated Analog DDFS



(d). Music Synthesizer

Figure 2. DDFS block diagrams

4.1 Digital DDFS

A *digital DDFS* circuit consists of a *phase register* and an adder and its block diagram is shown in Figure 2(a). The output is the MSB (most significant bit) of the register, which is a square wave, and the input is the *frequency control word*, labeled *M* in the diagram, whose value is added to the phase register in every clock cycle.

Let f_{sys} and f_{out} be the frequency of the system clock and the frequency of the output square wave, let *N* be the number of bits in the phase register and the adder, and let *M* be the value of the frequency control word. The typical width of *N* is between 24 and 48 bits.

The value of the phase register starts from 0, gradually increases to 2^N-1, and then wraps around. The MSB starts as 0, changes to 1 when the phase register reaches 2^{N-1}. It then returns to 0 and repeats when the phase register wraps around. The duration of incrementing from 0 to 2^N-1 is the period of the MSB (i.e., $1/f_{out}$). Since *M* is added to the phase register every clock cycle, it requires $2^N/M$ additions to complete one circulation and the required time for this task is $(2^N/M) * (1/f_{sys})$; i.e.,

$$1/f_{out} = (2^N/M) * (1/f_{sys})$$

The DDFS system can generate the desired output frequency by properly setting the value of *M*:

$$M = (f_{out}/f_{sys}) * 2^N$$

Note that the shape of the digital DDFS is always a square wave and its amplitude cannot be adjusted. An additional circuit is needed to control the duration.

4.2 Analog DDFS

An *analog DDFS* consists of an additional *phase-to-amplitude lookup table*, a *DAC* (*digital-to-analog convertor*), and a *low-pass filter*, as shown in Figure 2(b). The key component is the lookup table. Let the *N*-bit output of the phase register be $p_{N-1}\ p_{N-2}\ p_{N-3} \ldots p_0$. The digital DDFS uses the MSB (i.e., p_{N-1}) as the square wave output. One interpretation is to treat the p_{N-1} bit as a signal that divides the output period into two equal parts (i.e., two equal *phases*). The values of 0 and 1 are assigned to the two phases, respectively. It is possible to assign multi-bit values, such as 2 and 7, for the amplitude, and the output wave will oscillate between 2 and 7 instead.

Similarly, if two MSBs (i.e., $p_{N-1}\ p_{N-2}$) are considered, the same period is divided into four phases. Different values can be assigned to the four phases. The same concept can be extended to *S* MSBs, which leads to 2^S phases in a period. A phase-to-amplitude lookup table with 2^S entries can be created to define the shape of the output waveform. To obtain a sinusoidal wave, we just need to create a lookup table with the sinusoidal function. The output of the lookup table is the *digitized* sinusoidal wave. The subsequent DAC transforms it into a continuous analog signal and the low-pass filter removes the high-frequency noises.

A DAC is usually implemented by an analog circuit in a separate IC chip. However, because the frequency of the system clock (around 50 MHz to 100 MHz) is much higher than that of the

audio signal, it is possible to "oversample" the output signal and use a *one-bit sigma-delta DAC*[14,15]. The one-bit DAC can be implemented digitally without any analog component and can perform *noise shaping*, which pushes quantization noises into the high-frequency range. This imposes less constraint on the subsequent low-pass filter and a simple RC filter can be used.

As in digital DDFS scheme, only the frequency aspect of the waveform can be adjusted and an additional circuit is needed to control the duration.

4.3 Modulated analog DDFS

The analog DDFS outputs a pure sinusoidal wave. We can *modulate* it to obtain more interesting effects. In signal processing, *modulation* is the process that modifies a high-frequency *carrier signal* in accordance with a low-frequency *message signal*. The carrier signal is a sinusoidal waveform and the message signal adjusts its amplitude, frequency, phase, or a combination of them. Assume that the carrier signal is $sin(2\pi ft)$. The modulated signals become the following:

- Amplitude modulation: $A(t)*sin(2\pi ft)$.
- Frequency modulation: $sin(2\pi(f+\Delta f(t))t)$.
- Phase modulation: $sin(2\pi ft+\Delta P(t))$.

The $A(t)$, $\Delta f(t)$, and $\Delta P(t)$ terms are slow time-varying message signals. The three schemes can be combined and the modulated signal becomes $A(t)*sin(2\pi(f+\Delta f(t))t+\Delta P(t))$. A DDFS system can incorporate the desired modulation schemes by inserting additional adders to adjust the frequency control value and the phase count and a multiplier to scale the amplitude. The complete diagram of a *modulated analog DDFS* system is shown in Figure 2(c).

A modulated analog DDFS system can control the frequency and the amplitude. It also can generate different wave shapes by loading "shape functions" to the phase-to-amplitude lookup table. However, the shape from a real music instrument is very complex and unstructured and thus the sinusoidal function is used in general.

4.4 Music synthesizer

An ADSR envelope generator circuit produces an amplitude envelope. The ADSR envelope contour is specified by seven parameters, including three amplitude points, A_1, A_2 and A_3, and four segment intervals, T_a, T_d, T_s, and T_r, as shown in Figure 1. The circuit can be constructed with an FSM (finite state machine) in conjunction with an amplitude counter. The FSM contains four main states, *state_atk*, *state_dcy*, *state_sus*, and *state_rel*, which correspond to the four segments of the ADSR envelope. The amplitude counter increments or decrements a specific amount in a state and the amount is derived from the designated time interval. For example, the increment amount in *state_atk* is $(A_1 - 0) / T_a$ and the decrement amount in *state_dcy* is $(A_1 - A_2) / T_d$. The counter is cleared to 0 initially. When the FSM is triggered, it enters *state_atk* and the counter increments with $(A_1 - 0) / T_a$ every clock cycle The FSM stays in *state_atk* until the counter reaches A_1. It then moves to *state_dcy* and the counter decrements with $(A_1 - A_2) / T_d$. When the counter reaches A_2, the FSM moves to *state_sus* and the counter decrements with $(A_2 - A_3) / T_s$. The same activity is repeated until the FSM completes *state_rel*. As the FSM progresses, the counter continues incrementing or decrementing a specific amount.

The counter output corresponds to the envelope contour in Figure 1. Note that the incrementing and decrementing amounts are calculated in advance and used as FSM inputs. No division circuit is needed.

The block diagram of a music synthesizer is shown in Figure 2(d). It is composed of a DDFS circuit and two ADSR envelope generators. The first one modulates the amplitude, as discussed in Section 3.2. The second one is optional. It is connected to the frequency modulation input to generate special effects.

A music synthesizer uses the DDFS circuit to specify the frequency of a music note and uses the ADSR envelope generator to control its amplitude and duration as well as to manipulate the shape of the envelope.

4.5 Music synthesizer I/O core

The music synthesizer discussed in the previous subsection is constructed from scratch with hardware. As the circuit becomes more sophisticated, the number of input signals increases. It will be tedious to set up the system parameters and play a music melody with just buttons and switches. A better alternative is to use software to perform this task. Since an ADSR envelope corresponds to the duration of a music note, its interval is around a fraction of a second. Setting up and initiating envelopes can be easily handled by an embedded processor or microcontroller.

To connect the music synthesizer to a processor bus, the music synthesizer should be wrapped with additional interface logic and appeared as an I/O core (i.e., an I/O peripheral). A common interface scheme is *memory-mapped-I/O*, in which an I/O core is treated as a collection of addressable registers and the processor uses the normal memory read and write instructions to access an I/O register[13]. The write interface with four addressable registers is shown in Figure 3. The decoding circuit uses two lower address bits and the *chip-select* (*cs*) signal to generate a register enable signal. When attached to a processor bus, the peripheral is assigned with a *base address*. When the processor writes a register, the system interface decodes the upper address bits and asserts the *cs* signal, and the local write interface decodes the two lower address bits and stores data to the designated register.

A similar write interface can be created for the music synthesizer to make it a standard I/O core. Since the memory-mapped-I/O scheme is used in FPGA's internal bus interconnect, the music synthesizer core can be easily incorporated into an FPGA based embedded system.

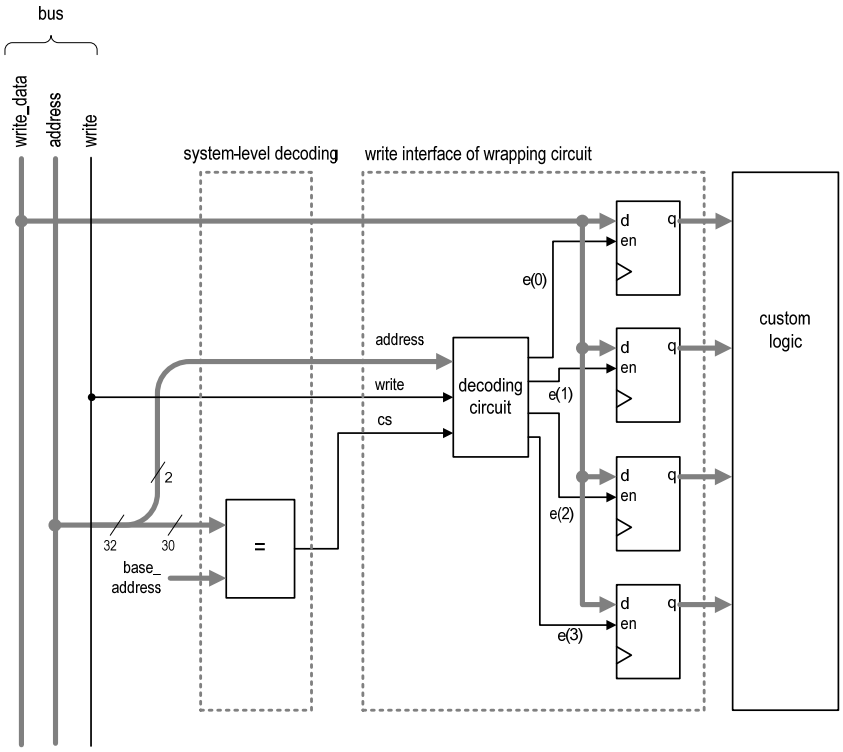


Figure 3. Memory-mapped I/O write interface

5. Lab Integration

5.1 Introduction

The purpose of this work is not to create a sophisticated project for a single course but to develop a collection of theme-based experiments and projects for the entire computer engineering curriculum. We divide the lab experiments and projects into following levels:

- Level 1: Freshman engineering.
- Level 2: Basic digital systems.
- Level 3: Advanced digital systems without processor.
- Level 4: Advanced digital systems with processor.
- Level 5: Capstone projects.

Level 1 is intended for freshman engineering students. Many schools now have an "introduction to engineering'' course for new engineering students. It is usually a project-oriented course to introduce the basic engineering concepts and practices.

Level 2 corresponds to the basic digital systems topics in the curriculum[16], which cover combinational circuits, sequential circuits, and FSM. After the basic materials, there is no single "standard" follow-up subject. The advanced topics can be spread over a wide variety of courses, such as advanced digital systems, computer organization, VHDL/Verilog, embedded systems,

hardware-software co-design, and so on. For our development purposes, one key distinction is whether a processor is incorporated into the course. Based on this, we divide the follow-up into two levels – Level 3 (*without processor)* and Level 4 (*with processor*). Because the course length (e.g., semester versus quarter) and the credit hours (e.g., 3- versus 4-credit hours) are different in each curriculum, a course may contain more than one level.

The level 5 is to apply the materials from the previous levels for the term projects or capstone design projects.

5.2 Lab logistics and adoption

The experiments and projects require a simple microcontroller board, an FPGA prototyping board, and a powered speaker. The microcontroller board is for the introductory freshman lab and the FPGA board is for the remaining labs. We select the Arduino board for the microcontroller board. It contains an 8-bit microcontroller and a dozen connectors to access external I/O devices. It supports a subset of C/C++ and provides a simple user development environment[17]. The setup is targeted for beginners without much prior programming or hardware experience. There are many FPGA boards designed for academic learning, such as the Altera DE series boards and Xilinx Nexys and Basys series boards[18,19]. Since the DDFS system does not use any proprietary vendor IP (intellectual property) and the music synthesizer requires only one output pin, they can be implemented in any entry-level FPGA board.

The experiments and projects are intended as companion hands-on exercises for the theoretical topics covered in a typical computer engineering curriculum. Adopting and integrating the experiments into an existing curriculum are straightforward and flexible. There is no need to modify the existing course structure since only the lab portion is updated. An instructor first identifies the corresponding experiment of a specific subject area and then substitutes it with a sound-theme experiment. To achieve the best result, the instructors in charge of various courses should coordinate and perform this across the entire curriculum.

5.3 Level 1 experiments

The freshman engineering course is usually a project-oriented course to introduce basic computer engineering concepts and practices. It assumes that the students only have high school math and science. The experiments in this level utilize software generated square wave, which is obtained by turning on and off an output port at a specific interval. The Arduino's built-in function, *tone( )*, can also be used for this purpose. Following are some basic experiments:

1. Two-tone police siren. The British police car siren produces a two-tone sound, in which the 440 Hz and 550 Hz tones alternate every second. We can derive a program to mimic the sound of the siren. The siren tone can be considered as frequency modulation resembling the BFSK (binary frequency-shift keying) scheme.
2. Sweep police siren. The US police car siren produces a sweeping frequency between 635 Hz and 912 Hz. We can derive a program to mimic the sound of the siren. This also introduces the basic concept of frequency modulation.
3. Music note generation. We can follow the discussion in Section 2 to calculate the frequency of a music note and generate the tone.

4. Melody player (part I). A *music melody* contains a sequence of notes. We can derive a program to play a music melody. Initially, we assume that the duration of the notes is fixed.
5. Melody player (part II). We enhance the previous program to include the duration information.
6. IR (Infrared) remote control. Remote control uses an IR signal to send commands. A common scheme is to transmit a 38 KHz square wave for the "on" period and 0 for the "off" period. The signal can be generated similar to a music tone. We can use an IR LED and develop a program for IR remote control.

5.4 Level 2 experiments

Level 2 corresponds to the basic digital systems topics and involves the digital DDFS system and the analog DDFS system. The former is related to sequential circuits and counters and the latter involves several more advanced topics, including the lookup table, the ROM, and the digital-to-analog conversion. Note that the digital DDFS system is essentially the hardware implementation of the software *tone( )* function in Level 1 and the analog DDFS system further enhances the functionality to generate a better analog sinusoidal wave.

The experiments in this level are based on the two DDFS systems. Following are some experiments:

1. Two-tone police siren. This repeats Experiment 1 in Level 1 but uses pure hardware implementation. The system can be done with two DDFS circuits and a multiplexer, as shown in Figure 4. One DDFS circuit generates the one-second interval (1 Hz signal) to control the multiplexer. The other DDFS circuit generates the tones.
2. Sweep police siren. This repeats the Experiment 2 in Level 1 but uses pure hardware implementation. "Sweeping frequency" implies to update the frequency control word of the DDFS circuit gradually. This can be accomplished with a counter and a decoding circuit.
3. Music note generation. This repeats Experiment 3 in Level 1 but uses pure hardware implementation. The implementation follows the basic approach discussed in Section 2. We first determine the frequency control word values for the 12 notes in the octave 0 and then use a barrel shifter (to perform $*2^n$) to obtain the values in the octave n.
4. Note duration control. A music note contains duration information, which specifies the time interval in which the tone is to be played. A digital *monostable multivibrator*, which generates a single pulse of a specified width after a trigger, can be used to gate the DDFS output via a multiplexer, as shown in Figure 5.
5. General-purpose wave generator. The analog DDFS system can be expanded to generate the triangular wave and the ramp wave. This can be done with the clever manipulation of the phase register output.
6. IR remote control. This repeats Experiment 6 in Level 1 but uses pure hardware implementation. The multiplexing configuration similar to that in Figure 5 can be used to control the on and off periods of the IR transmitter.
7. Quadrature phase carrier generation. In addition to the main carrier signal, some communication schemes require an additional 90 degree out-of-phase signal, known as

the *quadrature* component. It implies that $sin(2\pi ft)$ and $cos(2\pi ft)$ waveforms need be generated at the same time. We can configure the phase-to-amplitude lookup table with a dual-port memory module to support simultaneous access.

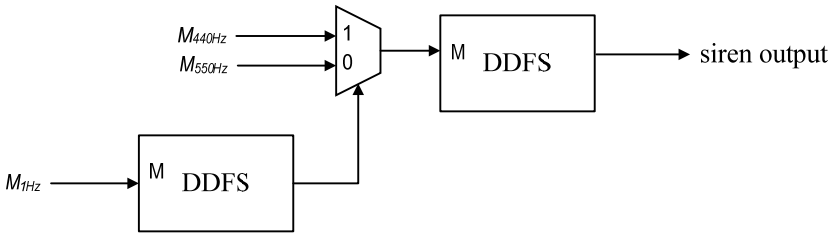


Figure 4. Block digram of two-tone siren

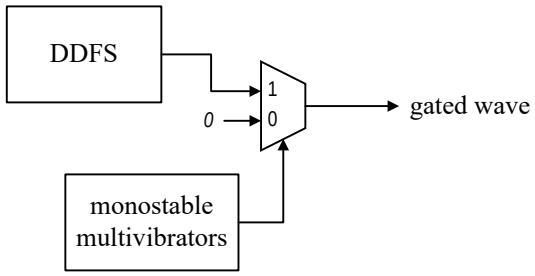


Figure 5. Block digram of tone duration control

Note that the melody player (Experiments 4 and 5) in Level 1 is not included. While it is straightforward for software to play notes with a *for* loop, this type of sequential execution requires substantial work for the hardware implementation. They are left for the next level.

5.5 Level 3 experiments

Level 3 corresponds to the advanced digital systems topics and involves the modulated analog DDFS system and the music synthesizer. They are related to more advanced topics, such as binary multiplier, fixed-point arithmetic, and FSMD (FSM with a data path)[13].

The experiments in this level are based on the music synthesizer. Following are some experiments:

1. Melody player (part I). This repeats Experiment 4 in Level 1 but uses pure hardware implementation. A custom RAM based FIFO (first-in-first-out) buffer can be used for storage. The melody data will be stored into the RAM as the initial values. An FSMD will control the overall operation. It will read the note's frequency from FIFO's head, write it back to the FIFO's tail (to have the melody played continuously), send it to the DDFS, wait for a fixed amount of time, and then repeat.
2. Melody player (part II). This repeats Experiment 5 in Level 1 but uses pure hardware implementation. This extends the previous experiment by augmenting an additional

duration field in the FIFO entry. The FSMD will use this duration information to time the FIFO retrieval.

3. Push-button piano player. This extends Experiment 3 in Level 2. We can use seven pushbuttons for the primary notes. Pressing a button will trigger the ADSR envelope generator to produce an envelope. Additional switches can be used to select octaves and pre-defined ADSR envelopes.
4. Push-button piano recorder. This combines Experiments 1 and 3. We can record the button sequences in the FIFO buffer and then play it back as a melody. The implementation requires a comprehensive FSMD that controls the recording and playback operation.
5. Additive harmonic synthesis. A harmonic is a signal whose frequency is an integer multiple of the fundamental frequency. We can expand the DDFS core to allow the addition of three harmonics to produce more interesting tones.
6. DAC interface. While one-bit sigma-delta ADC is adequate for an audio signal, an external DAC device is needed to produce a high-frequency sinusoidal wave. These DAC devices usually contain an SPI or I^2C interface. We can first design an SPI or I^2C controller and then create a top-level FSMD that controls the ADC operation and streams the DDFS output.

5.6 Level 4 experiments

Level 4 includes an embedded processor and involves the music synthesizer I/O core. The emphasis is on hardware-software interface and hardware acceleration. Following are some experiments:

1. *Tone( )* function implementation. A pre-designed *tone( )* function is provided in Arduino platform. We can implement the same function with the hard-core or soft-core processor of the FPGA board.
2. *Tone( )* function analysis. For the *tone( )* function of Experiment 1, we can analyze and profile the code to obtain the CPU utilization of different frequencies and to determine the maximal frequency that can be obtained by this method.
3. DDFS analysis. We can analyze the digital DDFS system to determine the maximal frequency that can be obtained by this method.
4. Melody player. This repeats Experiment 2 in Level 3. We can develop a software program to play a melody using the synthesizer I/O core. The software essentially replaces the FSMD controller and the FIFO buffer.
5. Push-button piano recorder. This repeats Experiment 3 in Level 3. We can use processor's general-purpose I/O core to access the buttons and the switches and develop a software program to control and coordinate the I/O operation.
6. DAC interface. This repeats Experiment 6 in Level 3. We can use a software program to configure the DAC device and to coordinate and control the DDFS output streams.

5.7 Level 5 experiments

Level 5 is for the term projects or capstone design projects. Because the materials in the previous levels cover custom hardware, general embedded systems, and hardware-software co-design, there are plenty opportunities for interesting and challenging projects. The projects can

enhance the synthesizer or integrate it with other peripherals, such as a touch screen, a TFT (thin-film-transistor) display, and sensors. Following are some examples:

1. Enhanced DDFS. DDFS is the main scheme to produce high-frequency sinusoidal wave in a communication system. Advance techniques, such as fast adder, pipeline structure, etc., can be used to improve the performance of the DDFS system and to achieve a higher operating frequency.
2. Sound processor. The early computer or video game console used a *sound chip*, such as Yamaha YM3812, to produce sound. The music synthesizer core can be extended to emulate this type of devices.
3. Function generator. A function generator is a piece of equipment that can generate simple repetitive waveforms. The frequency and the amplitude can be adjusted. The shape of the waveform is limited to a sinusoidal wave, a square wave, a triangular wave, and a ramp wave.
4. Arbitrary waveform generator. Arbitrary waveform generator extends the capability of the function generator by allowing a user to define the shape of the waveform. This can be achieved by incorporating a write interface into the phase-to-amplitude lookup table.
5. Sample based synthesis. This scheme creates better sound by recording a sample from a real music instrument, digitizing the sample, and storing the data points to the phase-to-amplitude lookup table.
6. Innovative music instrument. A variety of input devices and sensors, such as a keyboard, a touchpad, a force sensor, a proximity sensor, an accelerometer, etc., can be used as user interface to control and adjust the frequency, the amplitude, and the shape of the DDFS output.
7. Special effect circuit. The DDFS output can be further processed to produce special sound effects. Possible extensions include a tunable low-pass filter, a circuit to generate echoes, a circuit to generate reverberations, etc.
8. Visual GUI. A GUI (graphic user interface) with a TFT liquid-crystal display or a smart phone can be created as the control panel for above projects.

5.8 Evaluation

The sound theme is part of the effort to create a "spiral lab framework"[20]. Its effectiveness is evaluated by an array of assessment instruments, including contents tests, lab works, student survey, and student interviews. The data collection is in progress. After the completion, comparisons will be made between gain scores of each class, survey and interview differences, as well as any differences in available formative course assessments, such as student homework and participation. If the sample is diverse enough, we will also examine which curriculum is more effective for high- vs. low-achieving students, as well as differences in the effectiveness of the curriculum based on gender and other demographic factors if available.

6. Summary

We follow the "spiral" model recommended from a recent study and develop a continuous and coherent series of sound-theme based experiments and projects for computer engineering curriculum. They connect and integrate the individual courses through a cohesive lab framework. The labs spread over all hardware related courses, including freshman engineering, introductory

digital systems, advanced digital systems, computer organization, embedded systems, and hardware-software co-design. The complexities and abstraction levels of experiments and projects gradually grow as students progress through the curriculum. Key concepts are repeated in different courses with increasing sophistication and studied from different aspects and contexts. The experiments and projects can be realized in simple microcontroller and FPGA boards and can be easily incorporated into any existing curriculum.

7. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1504030.

Bibliography

[1]. S. A. Ambrose et al., *How Learning Works: Seven Research-Based Principles for Smart Teaching*. Jossey-Bass, 2010.
[2]. C. J. Atman, et al., *Enabling Engineering Student Success: The Final Report for the Center for the Advancement of Engineering Education*, 2010.
[3]. S. Sheppard, et al., *Educating Engineers: Designing for the Future of the Field*. Jossey-Bass, 2009.
[4]. T. A. Litzinger, et al., "Engineering Education and the Development of Expertise," *Journal of Engineering Education*, pp.123-150, January 2011.
[5]. J. E. Froyd and M. W. Ohland, "Integrated Engineering Curricula," *Journal of Engineering Education*, pp.147-164, January 2005.
[6]. Wikipedia web site https://en.wikipedia.org/wiki/Synthesizer.
[7]. Wikipedia web site https://en.wikipedia.org/wiki/Musical_tone.
[8]. Wikipedia web site https://en.wikipedia.org/wiki/Musical_note.
[9]. C. Roads et al., *The Computer Music Tutorial*, MIT Press, 1996.
[10]. Wikipedia web site https://en.wikipedia.org/wiki/Synthesizer#ADSR_envelope.
[11]. M. Genovese, et al., "Analysis and comparison of Direct Digital Frequency Synthesizers implemented on FPGA," *Integr. VLSI Journal*, March 2014.
[12]. J. Vankka, *Direct Digital Synthesizers: Theory, Design and Applications*, Springer, 2001.
[13]. Pong P. Chu, *Embedded SoPC Design with Nios II Processor and VHDL Examples*, John Wiley & Sons, Inc., 2011.
[14]. R. Schreier and G. C. Temes, *Understanding Delta-Sigma Data Converters*, Wiley-IEEE Press, 2004.
[15]. S. R. Norseworthy, R. Schreier and G. C. Temes, *Delta-Sigma Data Converters: Theory, Design, and Simulation*, Wiley Interscience, 1997.
[16]. IEEE and ACM Joint Task Force on Computer Engineering Curricula, "Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering," 2004.
[17]. S. Monk, *Programming Arduino: Getting Started with Sketches*, McGraw-Hill, 2011.
[18]. Terasic web site http://www.terasic.com.tw/en/.
[19]. Digilent web site https://store.digilentinc.com/fpga-programmable-logic/.
[20]. Pong P. Chu, Chansu Yu, and Karla Mansour, "A Spiral Computer Engineering Lab Framework," accepted for presentation *in AAAS/NSF Symposium on Envisioning the Future of Undergraduate STEM Education: Research and Practice*, 2016.