DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 311 – Digital Systems Design
2015/2016 Term 1

Lab 5: Sound, Timers, and Memory

Working week: November 9-13, 2015
Marking week: November 16-20, 2015

In this lab, you will use the sound output on your DE2 board. Your circuit will synthesize sounds that can be played through a earphones (or a speaker). In doing so, you will get practice implementing interacting counters, simple arithmetic, and will be introduced to the use of embedded memory blocks on an FPGA. You will also learn about how sound can be represented in the digital domain.

The top level block diagram of the system you will create is shown in Figure 1. The block labeled "Your Circuit" is what you will design. The audio_codec, the audio_and_video_config, and the clock_generator blocks will be given to you. The interface between your circuit and the audio codec is simple and will be described below. Note that your circuit will be clocked by the 50MHz clock and will receive input from 7 slider switches (SW 6..0).

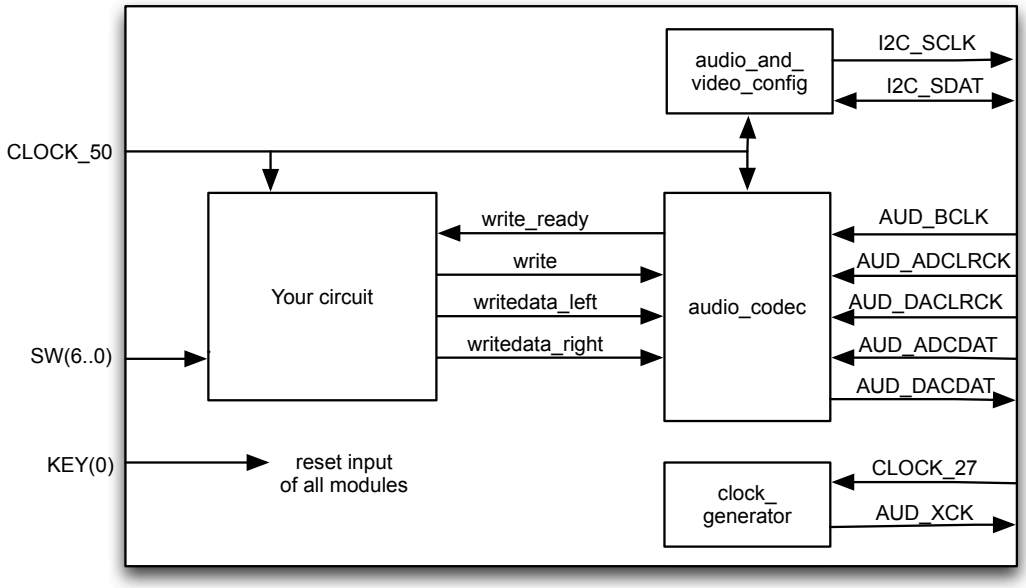


Figure 1: Top Level Block Diagram of your System

Task 1: Learning how sound is represented in the digital domain and our CODEC

a*) Sound in the digital domain:*
Before building this circuit, you need to understand how sound is represented in the digital domain. Some of this discussion and the figures are taken from lectures at the University of Toronto (for a similar course).

A sound is transmitted as a wave. The frequency of the wave represents the frequency (tone) of the sound, and the height of the wave (above and below the centre) determines the volume of the sound. Humans typically hear sounds from between about 20Hz to 20KHz. On a piano, Middle C is about 262 Hz. Thus, if you want to generate a Middle C tone, you would create a wave of 262 Hz; the amplitude of the wave would determine the volume.

On a piano, keys typically have a repeating pattern (A, B, C, D, E, F, G) [you may also know the tones as (do, re, me, fa, so, la, ti do)]. This pattern repeats; each repetition is called an octave. Two notes are an octave apart if the higher note is double the frequency of the lower note. The concept of an octave is shown graphically in Figure 2.

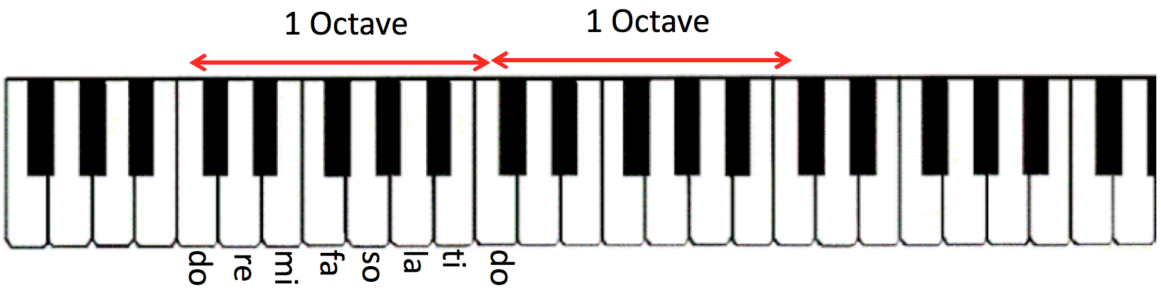


Figure 2: Piano keyboard showing octaves. Each octave, the frequency doubles

In the digital domain, waves are represented by a set of samples, as shown in Figure 3. In this example, a collection of values, some negative and some positive can be combined to "look like" a sine wave. Note that the frequency of the samples is much higher than the frequency of the sine wave. In our system, we will be using 48,000 samples per second to describe waveforms that range from roughly 262 Hz to 523 Hz.

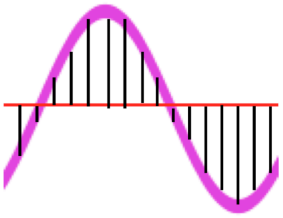


Figure 3: Sampling a sinusoidal waveform

Question to make sure you understand: assuming a sampling rate of 48,000 samples per second, how many samples are there per wavelength of a 262 Hz sound? The answer is 48,000 / 262 = 183.

Although the height of each sample could be calculated using a properly scaled sine function, it is often easier to approximate the sine wave as a square wave (it will sound slightly different, but the note is still recognizable). This is shown graphically in Figure 4. The reason you might want to do this is that it makes computing the magnitude of each sample much easier. For example, as in the above example, a wave that sounds like Middle C has 168 samples. Using a square-wave approximation, the first 84 samples can be a very negative number and the next 84 samples can be a very positive number. The magnitude of the negative and positive samples represent the volume of the sound. This is shown graphically in Figure 5 which shows two different frequencies represented by square wave samples.

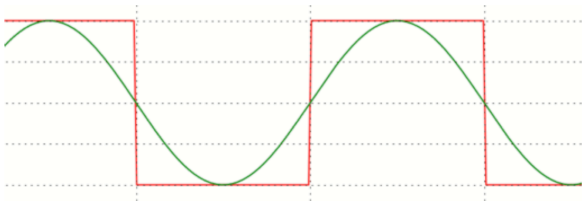
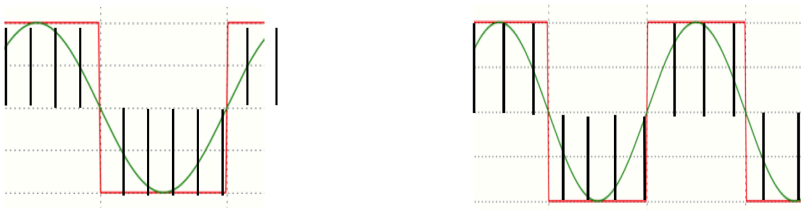


Figure 4: Approximating a sine wave with a square wave



a) Low Sound b) High Sound

Figure 5: Sampling square waves

Sound can be combined by adding the waveforms. If you add a wave of 262 Hz to another wave of 294 Hz, the result will sound as if note C and D are being played simultaneously. You can add as many notes as you like this way. In the digital representation described above, this means adding the samples from each waveform to produce a new waveform.

b) Our CODEC:

We are supplying you with a pre-designed CODEC. This circuit was designed and distributed by Altera. As shown in Figure 1, the CODEC actually consists of three blocks; you will include these three blocks in your VHDL code (we will provide a stub with the component declarations already there for you).

The operation of the CODEC is shown in Figure 6. Your circuit will insert samples into the head of two 128-entry FIFO queues, one dedicated to the right audio output and one dedicated to the left audio output. Your circuit can insert samples into the FIFO as long as there is space (details on this timing is below). Every 1/48000'th of a second, the CODEC will remove the top element in each FIFO and send it to the two speakers. The purpose of the FIFOs is to "balance out" bursty activity in your circuit. Without the FIFOs, your circuit would have to produce new samples exactly once per 1/48000'th of a second, which may make timing in your circuit tricky. With the FIFOs, as long as the *average* rate of queue entry is the same as the average rate of queue exit (48000 samples per second), the system will correctly play notes. Note that the FIFOs are part of the CODEC core; you do not implement these FIFOs in your own circuit.

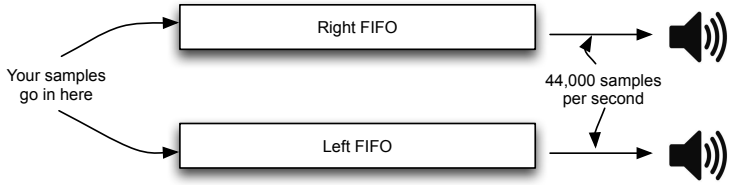


Figure 6: Structure of the FIFOs

To insert samples into the two FIFOs, your circuit should monitor the **write_ready** signal (which is an output of the CODEC). When this signal goes high, it means there is space in each FIFO for a sample. When your circuit sees it go high, it can drive the **writedata_left** and **writedata_right** buses with the two samples (one for the left speaker and one for the right speaker) and assert the **write** line (these are all inputs to the CODEC). You should then wait for **write_ready** to go low (this may take one or more cycles before it happens), and when it does, you can lower **write**, and start again.

The **writedata_left** and **writedata_right** buses are each 24 bits wide in our CODEC (it can be configured differently, but you will not change that). The smallest sample value is -2^{23} and the largest sample value is $2^{23}-1$. Therefore, if you want to make a maximum-amplitude square wave of 262 Hz, you would insert 91 samples of value -2^{23} followed by 91 samples of value $2^{23}-1$. Note that you can choose a smaller amplitude square-wave; the amplitude determines the volume of the sound. You will need to experiment to figure out what amplitude is reasonable with your earphones (I found that an amplitude of 2^{16} was plenty loud enough for me). You may want to make the maximum amplitude a **constant** in your VHDL code, so you can easily change it .

Note that the CODEC we are supplying also has the ability to operate in the other direction: taking samples from a microphone and supply them to your circuit. You won't be using that feature in this lab. To turn this feature off, you should keep **read_s** at 0, and ignore outputs **read_ready, readdata_left** and **readdata_right**.

Task 2: Playing a Single Tone (3 marks)

Starting with the stub file you can download from the Connect site, create a circuit that plays a single tone (Middle C, which is 262 Hz). Your circuit will be a state machine, clocked with a 50Mhz clock, that regularly samples the **write_ready** signal, and when it is true, sends a sample as described above.

Note that when compiling, you should use an SDC file which is included in the distribution zip file on Connect. This file is called **sound.sdc** and should be added to your project. The reason you need this is that it guides the optimization of the phase-locked loops that are part of the CODEC core. If you add it to your project, and give it the same name as your top level, then it will automatically be used when you compile.

To test your circuit, you will need earphones plugged into the line-out port of your DE2. For hygiene reasons, it is recommended that you use your own earphones. If you do not have earphones, and are not able to buy any, there will be some available in the CPEN 311 lab that you can borrow during your scheduled lab period. You can also use speakers, but to avoid annoying others, please do not do this while debugging in the lab.

To figure out if you are playing middle C, there are plenty of web sites or Apps (I use "Tone Generator" for my iPhone) that can generate a tone of a known frequency.

Assuming you get Task 3 working, you will not need to demonstrate Task 2. However, you may want to keep it around in case you run into problems in Task 3 and want to demonstrate *something*.

Task 3: One Octave Piano – Multiple Tones (6 marks)

In this task, you will create a piano that covers one octave. The keys will be implemented using switches SW6 downto SW0. SW6 should be Middle C, SW5 should be D, SW4 should be E, etc (you do not need to support sharps and flats). You will need to look up (Google) the frequencies for each note (there is a complex formula to work out each frequency, but you will probably find it easier to hardcode the 7 frequencies using a constant array). The operation of your circuit will be: if a switch is put into its "up" position, the corresponding note is played. If the switch is put into its "down" position, the corresponding note is *not* played.

If more than one switch is in the "up" position, you should play all notes corresponding to "up" switches simultaneously. To play multiple notes at the same time, you superimpose the samples as described earlier. You will need to make sure that the amplitude of the resulting waveform does not overflow the number of bits in our DAC (24 bits).

Be sure to use the SDC file as in Task 2. If you renamed your top level, you will need to rename the SDC file.

In this task, depending on how you write your code, you may run into a problem you have not had to deal with before. If you are not careful, it is possible that the critical path of your circuit is longer than the period of the 50Mhz clock. As you know from class, if this occurs, your circuit will not work properly. You may find this is an issue if you use a lot of "mod", division, or scaling operations. The SDC file you are using, a clock constraint for the CLOCK_50 clock is specified (you saw this in Slide Set 10). However, it is very possible that, during compilation, the tool is unable to find an implementation that meets the timing constraint. To determine the maximum speed at which your circuit can run, after compiling, open "TimeQuest Timing Analyzer" from the "Table of Contents" window to the left of your main Quartus II window (you can also run the timing analyzer from the Tools menu, although that interface will be more cumbersome to deal with). Within "TimeQuest Timing Analyzer", open "Slow Model" and then "FMAX summary". You should see something like Figure 7. In this case, the maximum clock frequency the circuit can run at is 52.16 MHz. Since this is faster than 50 Mhz, we are fine. If you find that this number is lower than 50Mhz, then you have a problem.

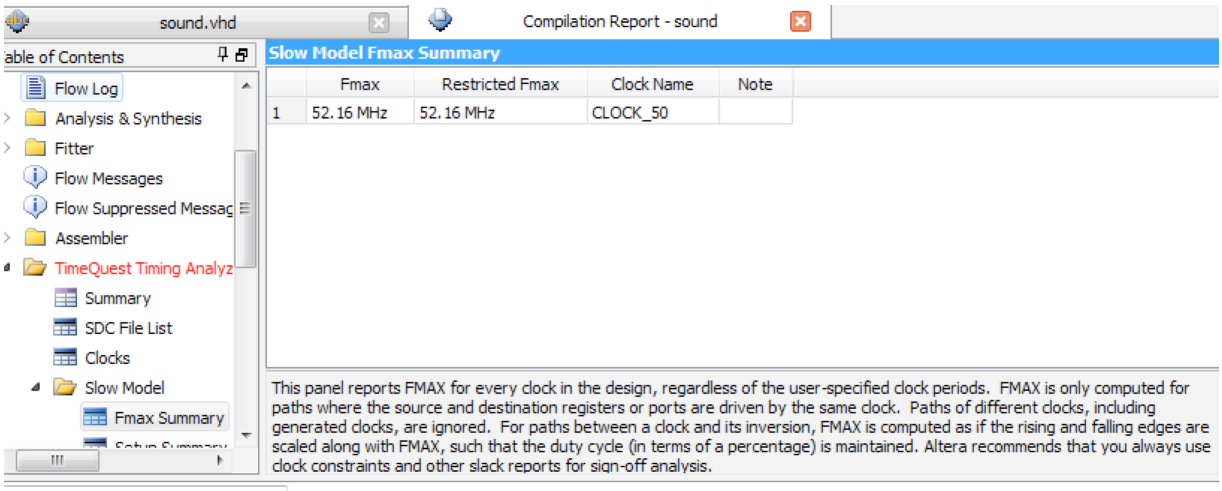


Figure 7: Checking your clock speed. Make sure it is >50Mhz

If your clock speed is less than 50Mhz, there are several things you can do. Check the structure of your code and try to remove slow operations like mod, division, or multiplications. You can see the offending paths if you choose "Worst Case Timing Paths" as shown in Slide Set 10, Page 16. There are lots of different ways to implement this circuit, some of which will result in much faster clock speeds than others. In my implementation, I was able to get more than 50 Mhz without too much trouble.

Challenge Task: (1 mark)

Challenge tasks are tasks that you should only perform if you have extra time, are keen, and want to show off a little bit. This challenge task is only worth 1 mark. If you don't demo the challenge task, the maximum score you can get on this lab is 9/10 (which is still an A+).

In this final task, you will interface an on-chip ROM (a ROM is a Read-Only Memory). The ROM will store a song, and your circuit will play the song (repeating when the song is over). Each entry in the ROM is a number 1 to 7, corresponding to notes C, D, E, F, G, A, B (in that order), ending in -1. Your circuit should read the memory, one line at a time, and play the appropriate note. Each note should be played for ½ a second.

In doing this, you should be aware of two requirements:

1. The ROM does not store samples. Instead, it stores notes, and your circuit will create the samples from the note. As described above, the ROM stores each note as a number from 1 to 7; each line in the ROM corresponds to a different note.
2. The duration of each note should be the same, regardless of the frequency. It would be incorrect if the duration of a A was shorter than a C.

To complete this challenge task, you'll have to read about how to implement ROMs in Quartus II. It is not too difficult; you can find plenty of examples on-line using Google.

You can make the simplification that your circuit only plays one note at a time. If you really want a challenge, modify your circuit so the song could contain "chords" (up to 3 notes played at the same time).