Sound Generation

Purposes:

- Learn how to generate sounds using Verilog programming and the audio output on the FPGA board

References:

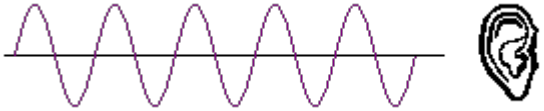
- Section 16 "Mono Audio Output" in the *NEXYS4-DDR Reference Manual*
- *http://www.howmusicworks.org/100/Sound-and-Music*

Special material needed for this lab:

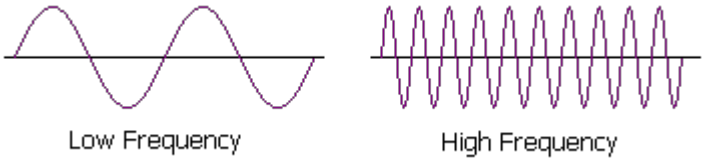
- Speaker

Sound and Music

Sound is the *vibration* of air particles, which travels to your ears from the vibration of the object making the sound. These vibrations of sound in the air are called *sound waves*. Musical sounds are vibrations that are strongly regular. When you hear a regular vibration, your ear detects the frequency, and you perceive this as *the pitch* of a musical tone.



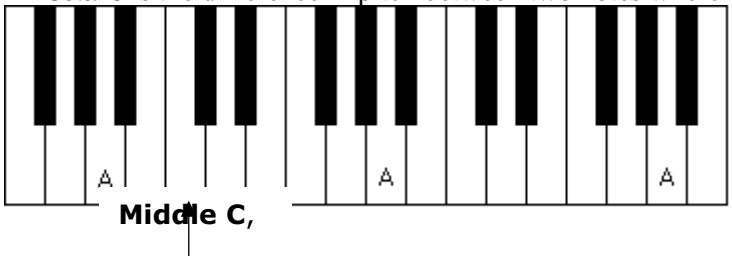
There are two main properties of a regular vibration - *the **amplitude*** and *the **frequency***. Amplitude is the size of the vibration, and this determines *how loud* the sound is. Frequency is the speed of the vibration, and this determines *the pitch* of the sound.



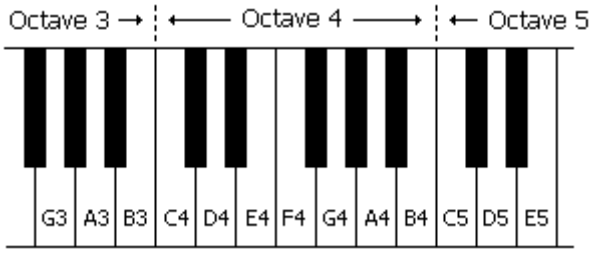
Frequency and pitch describe the same thing, but from different viewpoints. While frequency measures the cycle rate of the physical waveform, pitch is how high or low it sounds when you hear it. The higher the frequency of a waveform, the higher the pitch of the sound you hear. Human ears can only hear sounds within a certain range of frequencies. As people grow older, their hearing range reduces. A young person can usually hear sounds in the range of 20 Hz to 20,000 Hz.

Something very interesting happens when *you double the frequency* of a note. The pitch of the doubled frequency sounds higher, but somehow the same as the original note, while the pitches of all frequencies in between sound quite different. For example, the pitch of frequency *440 Hz* is the note ***A***, while the pitch of frequency 880 Hz is higher, but sounds like the same note.

An *octave* is the difference in pitch between two notes where one has twice the frequency of the other.

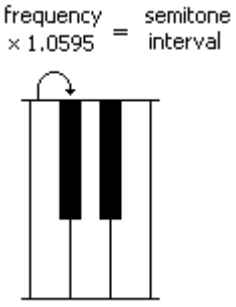


Two notes which are an octave apart always sound similar and have the same note name, while all of the notes in between sound distinctly different, and have other note names. This is a very important concept in music. *Notes* fall into *groups of twelve*.



There is a magic number in western music, known as the **twelfth root of two**, and it has a value of approximately *1.0595*. This is the number that, when multiplied by itself twelve times, gives a result of two. Why is this important to music?

Remember that with notes one octave apart, the higher note has *double the frequency* of the lower note. The range of frequencies in between is divided up into the twelve steps that give us all of our notes. The frequency of a note, when multiplied by the twelfth root of two (*e.g., 1.0595*), gives the frequency of the next note up. The difference in pitch between adjacent notes is called a *semitone*. After doing this for twelve notes, you end up with twice the frequency, which is the note one octave up from the starting note.



The set of all musical notes is called the Chromatic Scale, a name which comes from the Greek word *chrôma*, meaning color. In this sense, chromatic scale means 'notes of all colors'. Colors, in fact, are also made up from different frequencies, those of *light waves*. Because notes repeat in each octave, the term 'chromatic scale' is often used for just the twelve notes of an octave. This method of dividing the octave using the twelfth root of two is known as ***equal temperament tuning***, pioneered several centuries ago in the time of *JS Bach*. Since then, the music of the western world has been based on the notes of the Chromatic scale. Equal temperament tuning was a major breakthrough in the development of music.

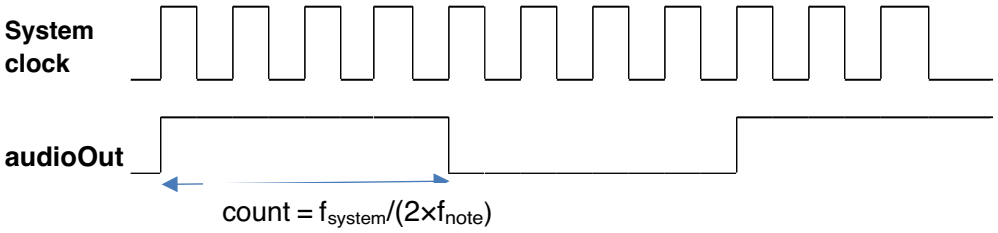
Chromatic Scale Notes
Frequencies of the twelve notes between note A at 440 Hz to note A one octave up

A	A#/Bb	B	C	C#/Db	D	D#/Eb	E	F	F#/Gb	G	G#/Ab	A
440.00	466.16	493.88	523.25	554.37	587.33	622.25	659.25	698.46	739.99	783.99	830.61	880.00

#: sharp (one semitone higher)
b: flat (one semitone lower)

Monotone Generation

Frequency



Notes	Frequency f_{note} (in Hz)	Count
A4	440	50_000_000/440
A4#		
B4		
C4		

Duration:

- Assume normal tempo of 120 bpm (beats/minute). That is, every beat lasts ½ second (**2** Hz).
- Normally, quarter notes last one beat.
- Hence, half notes last two beats, and whole notes last four beats.
- Eighth notes last half beats, and sixteenth notes last ¼ beats.

Binary representation of duration

Use a 5-bit binary number from whole notes to 1/16 notes

						Number of Beats	Hz
1	0	0	0	0	Whole note	4	
0	1	0	0	0	Half note	2	
0	0	1	0	0	Quarter note	1	2
0	0	0	1	0	Eighth note	1/2	
0	0	0	0	1		1/4	
0	1	1	0	0		3	
0	0	1	1	0		1 1/2	

```
module MusicSheet( input [9:0] number,
      output reg [19:0] note,//max 32 different musical notes
      output reg [4:0] duration);
parameter   QUARTER = 5'b00010;//2 Hz
parameter   HALF = 5'b00100;
parameter   ONE = 2* HALF;
parameter   TWO = 2* ONE;
parameter   FOUR = 2* TWO;
parameter C4=?, D4=?, E4 = ?, F4=?, G4 = ?,C5 = ?, SP = 1;

always @ (number) begin
case(number) //Row Row Row your boat
0:    begin note = C4;  duration = HALF;  end   //row
1:    begin note = SP; duration = HALF;   end   //
2:    begin note = C4; duration = HALF;   end   //row
3:    begin note = SP;  duration = HALF;  end   //
4:    begin note = C4; duration = HALF;   end   //row
5:    begin note = SP; duration = HALF;   end   //
6:    begin note = D4; duration = HALF;   end   //your
7:    begin note = E4; duration = HALF;   end   //boat
8:    begin note = SP; duration = HALF;   end   //
9:    begin note = E4; duration = HALF;   end   //gently
10:   begin note = SP; duration = HALF;   end   //
11:   begin note = D4; duration = HALF;   end   //down
12:   begin note = E4; duration = HALF;   end   //
13:   begin note = SP;  duration = HALF;  end   //
14:   begin note = F4; duration = HALF;   end   //the
15:   begin note = G4; duration = HALF;   end   //stream
16:   begin note = SP;  duration = HALF;  end   //

17:   begin note = C5; duration = HALF;   end   //merrily
18:   begin note = SP; duration = QUARTER;      end   //
19:   begin note = C5; duration = HALF;   end   //
20:   begin note = SP; duration = QUARTER;      end   //
21:   begin note = C5; duration = HALF;   end   //
22:   begin note = SP; duration = QUARTER;      end   //

23:   begin note = G4; duration = HALF;   end   //
24:   begin note = SP; duration = QUARTER;      end   //
25:   begin note = G4; duration = HALF;   end   //
26:   begin note = SP; duration = QUARTER;      end   //
27:   begin note = G4; duration = HALF;   end   //
28:   begin note = SP; duration = QUARTER;      end   //

29:   begin note = E4; duration = HALF;   end   //
30:   begin note = SP; duration = QUARTER;      end   //
31:   begin note = E4; duration = HALF;   end   //
32:   begin note = SP; duration = QUARTER;      end   //
33:   begin note = E4; duration = HALF;   end   //
34:   begin note = SP; duration = QUARTER;      end   //

35:   begin note = C4; duration = HALF;   end   //
36:   begin note = SP; duration = QUARTER;      end   //
37:   begin note = C4; duration = HALF;   end   //
38:   begin note = SP; duration = QUARTER;      end   //
39:   begin note = C4; duration = HALF;   end   //
40:   begin note = SP; duration = QUARTER;      end   //
```

```
41:   begin note = G4; duration = ONE;    end   //Life
42:   begin note = SP; duration = HALF;   end   //
43:   begin note = F4; duration = HALF;   end   //is
44:   begin note = E4; duration = HALF;   end   //but
45:   begin note = SP; duration = HALF;   end   //
46:   begin note = D4; duration = HALF;   end   //a
47:   begin note = C4; duration = HALF;   end   //dream
default:    begin note = C4; duration = FOUR;   end
endcase
end
endmodule
```

```
module SongPlayer( input clock, input reset, input playSound, output reg
audioOut, output wire aud_sd);
reg [19:0] counter;
reg [31:0] time1, noteTime;
reg [9:0] msec, number; //millisecond counter, and sequence number of musical
note.
wire [4:0] note, duration;
wire [19:0] notePeriod;
parameter clockFrequency = 100_000_000;

assign aud_sd = 1'b1;

MusicSheet  mysong(number, notePeriod, duration );
always @ (posedge clock)
  begin
      if(reset | ~playSound)
            begin
                counter <=0;
                time1<=0;
                number <=0;
                audioOut <=1;
            end
      else
            begin
                  counter <= counter + 1;
                  time1<= time1+1;
                  if( counter >= notePeriod)
                     begin
                        counter <=0;
                        audioOut <= ~audioOut ;
                     end      //toggle audio output
                  if( time1 >= noteTime)
                        begin
                              time1 <=0;
                              number <= number + 1;
                        end  //play next note
                  if(number == 48) number <=0; // Make the number reset at the end of the song
            end
  end

  always @(duration) noteTime = duration * clockFrequency / 16;
       //number of   FPGA clock periods in one note.
endmodule
```

Implementation

```
##PWM Audio Amplifier
set_property -dict { PACKAGE_PIN A11   IOSTANDARD LVCMOS33 } [get_ports { audioOut }];
#IO_L4N_T0_15 Sch=aud_pwm
set_property -dict { PACKAGE_PIN D12   IOSTANDARD LVCMOS33 } [get_ports { aud_sd }];
#IO_L6P_T0_15 Sch=aud_sd
```

Procedure:

1. Implement the example ‘song player’ and ‘music sheet’
2. Pick your own song and develop it and implement it on the FPGA board