THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE
Department of Electrical and Computer Engineering

CPE 324 Advanced Logic Design Laboratory
Laboratory Assignment #6
Digital to Analog Conversion using Pulse Density Modulation

(10% of Final Grade)

Purpose

In this laboratory you will use digital hardware to create a periodic analog signals of varing types by controling one of the FPGA's general purpose digital I/O pins. The Signal Processing, SP, algorithm you will use is called Pulse Density Modulation, PDM. You will first examine the use of Pulse Density Modulation to generate three different static waveforms on the DE2-115 trainer which you will verify for correctness in the laboratory using an oscilloscope and digital multimeter. You will then modify your design so that it will play back prerecorded audio through an amplified speaker that will be connected to the general purpose ditital I/O pins of the DE2-115 trainer that your are driving. You will then be asked to modify this design again so that it will generate an amplitude modulated analog Radio Frequency, RF, signal that is amplitude modulated by your audio. The frequency of the RF signal is to be 1 Mhz which will allow you to verify the functionality of your design by using a standard AM broadcast band receiver.

Verilog Modeling Constructs

This laboratory module utilizes the following aspects of the Verilog HDL language.

- Verilog **task** construct
- **signed** fixed-point data types
- Verilog **for** loop construct
- 2-Dimensional Arrays
    - declaration
    - initialization using individual assignments within an **initial** section
    - initialization using the **$readmemh** directive in an **initial** section
- Named procedural **always** blocks
    - Local variable declaration and usage
- **`define** directive

Pulse Density Modulation

Pulse density modulation is a common signal processing technique that can be used to create high-fidelity audio output in portable hand held devices without requring large amounts of analog circuitry. This form of modulation generates a binary bit stream in time, *b[0], b[1], ... b[n]*, where *b[i]={0,1}*, that causes digital voltage pulse values to occur in a sequence that when averaged corresponds closely to the targeted voltage of the analog signal that is being approximated. In Pulse Density Modulation, for each step in the binary stream, the output voltage can be represented as

$$V_{out}(i) = V_{max} \text{ when } b[i] = 1 \text{ or } V_{out}(i) = V_{min} \text{ when } b[i] = 0 .$$

(For the digital outputs on the Intel Cyclone IV FPGA of the DE2-115, this corresponds to $V_{max}= 2.5V$ and $V_{min}= 0V$.) If the pulse frequency of the binary bit stream is sufficiently large compared to the frequency of the desired analog signal, then connecting the digital output to a simple low-pass filter makes it possible to produce any voltage within the continuous V_{max} to V_{min} range. This voltage will simply reflect the moving average of positive pulses that have been produced by the PDM process. For example a bit stream of *b[i]=1*, for all values of *i*, will produce an output voltage of V_{max}. In a similar manner a bit

stream of *b[i]=0* for all *i* then the analog output voltage would correspond to V_{min}. An intermediate output voltage that would fall exactly in between V_{max} and V_{min} would be produced for bit streams that alternate evenly between *b[i]=0*, and *b[i]=1*.
Encoding the binary bit stream of the targeted analog signal using PDM utilizes the process of delta-sigma modulation. In PDM the analog signal is in effect passed through a one-bit quantizer. This quantizer produces an output bit of *b[i]=1* or a *b[i]=0* in the bit stream in a manner that is designed to reduce the quantization error which is the difference between the actual average digital output voltage level associated with the bit stream and the currently desired analog voltage level. The output of the quantizer is then negatively fed back in the process loop where it is then either added or subtracted from the quantization error which will in turn affect the threshold value that is used to quantinize the next bit that is produced. PDM has the effect of causing the average error to be distributed out across the bit stream which results in the analog voltage being closely approximated at any point in time by the bit stream average. More information on the Pulse Density Modulation, PDM, process can be obtained at
http://en.wikipedia.org/wiki/Pulse-density_modulation.

Figure 1 illustrates in both C/C++ and in Verilog HDL the case where a digital input signal, x, is multiplied together with a scale value to allow that signal's amplitude to be varied. (In the C/C++ verion it is assumed that the pdm function gets called once every sample period.)

Inputs:
x = desired value of base waveform at sample point.
scale = amplitude scale value
Output:
ret_val = binary output {0,1} at sample point.

```
// Pulse Density
// Modulation Function
// in C/C++

int pdm(float x,
   float scale) {
   int ret_val;
   static float ge=0;

   float x_total;
   x_total=x*scale;

   if (x_total>=ge) {
      ret_val = 1;
      ge = ge-x_total+1;
   }
   else {
     ret_val = 0;
     ge=ge-x_total-1;
   }
   return ret_val;
}
```

```
// Pulse Density Modulation Task in Verilog
task pdm(output d_out,input signed [1:-14] x,scale,
   inout signed [3:-28] ge);

   // 32-bit fixed point encoding of 1.0 assuming a
   // signed 3 bit mantisa and a 28 bit fraction
   `define ONE_32 {1'b0,3'b001,28'b0}
         // {sign bit, mantisa, fraction}

   // 32 bit fixed-point holder of x * scale data
   reg signed [3:-28] x_total;

   begin

   x_total = x*scale; // signal multiplication

   if (x_total >= ge)
      begin
      d_out = 1;
      ge = ge - x_total + `ONE_32;
      end
   else
      begin
      d_out = 0;
      ge = ge - x_total - `ONE_32;
      end
   end

endtask
```

Figure 1: Basic Pulse Width Modulation Algorithm
(C/C++ floating point and Verilog HDL Fixed Point Representations)

In both the C/C++ and Verilog cases, the ***x_total*** variable represents the current sample value after scaling and the ***ge*** variable is the running error varable which can take on a value in the range of -1 to 1. The range of the ***scale*** varible is from 0 to 1 which is designed to govern the relative strength of the resulting signal that is applied to the PDM process. If ***scale*** is at 0 then the resultant signal is at zero percent (zero output regardless of the signal). If ***scale*** has a value of 1 then the resultant signal is at the same value as the signal x. In other words the scale input allows the strength of the signal to be varied from zero percent, no signal, to 100 percent the entire signal is present. The actual signal to which the PDM method is being applied is the product of the original signal, ***x***, and this scale value times 2 which will be in the range of -1 to 1. n both cases, there are two inputs. One input is the reference signal that for purposes to be explained later is assumed to be within the range -0.5 to +0.5. The C/C++ implementation of this function incorporates 32-bit floating point mathematics while the Verilog version utilizes 32-bit integer mathematics where the values have been scaled to reflect a fixed-point representation. Both give about the same precision for the data ranges associated with this application.

Figure 2 illustrates the fixed-point interpretation of the signed input variables, ***x***, and ***scale***, and the internal variable ***x_total***. Because of its **signed** data type designation, the most significant bit of each of these variables is the sign bit and the number is represented in its twos-complement form. Declaring these input variable as **signed** means that comparisons such as greater than (>) and algebraic operators such as multiplication (*) will be performed in a manner where the most significant bit will be interpreted as the sign bit. Since the value of the quantity ***x*scale*** should always be between -1 and 1, the alpha sigma method employes both the addition or subtraction of the value 1 (as shown in Figure 1), which means that the value of the global error, **ge**, can at a given point in time be either positive or negative. This is the reason for using the **signed** data type. Fixed point values have the benefit that circuitry to implement arithmetic functions is easier to create and the amount of logic resources needed is much less that for a given floating point representation. The speed of fixed point operations is also significantly faster and much more predictable because there is no need to shift the mantisa to normalize the exponent.

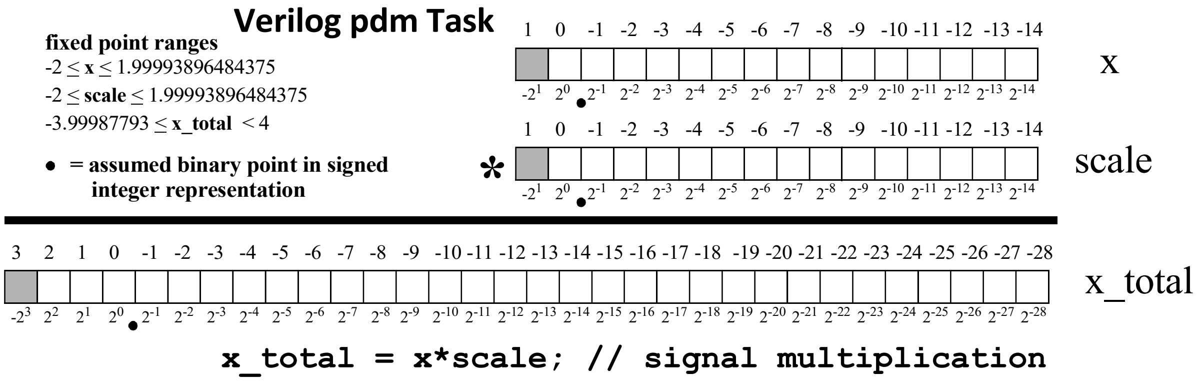
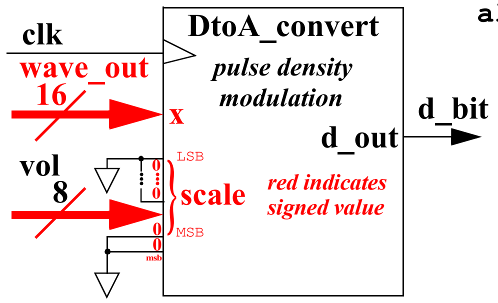


Figure 2: Signed Fixed-Point Range Analysis

Fixed-point assumes that there are a set number of binary places to the left and the right of a binary point. In fixed point the integer (mantisa) portion of the number is to the left of this point and the fractional component is to the right. In standard signed integers this binary point is always assumed to occur after the left-most least significant bit position in the number meaning that there is no fraction portion of the number only an integer portion. This is shown for the variables **x** and **scale** where bit positions -1 through -14 are considered to be to the right of the binary point and bit positions 1 and 0 are to the left of this point. Because the **signed** type is used bit 1, the MSB, will represent the sign bit. This assumed weighting makes it possible to approximate numbers in the range of -2 to just under 2 as shown in the figure. Thus the number 1.0 for inputs ***x*** or **scale** in Figure 2 would be represented in Verilog as {1'b0,1'b1,14'b0}. To accommodate the full range of possible signed fixed-point multiplication of ***x****scale*** the requires 32 bits, a one bit sign, three bit integer (mantisa) and a 28 bit fraction. This means that a value of 1.0 for ***x_total*** in Figure 2 is encoded in verilog as {1'b0,3'b001,28'b0}.

Figure 3 illustrates the case where only 8 bits of the scale variable are to be used in which case the lower order 6 bits are assumed to be zero and the sign bit and the integer bit is set to 0 with the **scale** input being placed in between allowing the signal **x** to be multiplied by the fractional component of the fixed point number. This was done to facilitate the DE2-115 case where there are only 8 dip switches that are available to vary the amplitude.

The *pdm* task can be called from within a procedural *always* statement as shown in Figure 3. Note that this procedural block has been named *DtoA_convert*. The global error term ***ge*** needs to be declared outside the pdm **task** because it will be overwritten each time the block is executed. This variable could be declared globally, outside the *DtoA_convert* always block, but since there is no other reason for other entities to access this it and data hiding is a desirable feature then it has been declared as a local variable to the *always* block. This is done by placing the declaration for ***ge*** variable after the **begin** statement in the block. Note that this variable is only visible within the named *always* block. It is also important to note that Verilog requires that the *always* block be named if one or more local variables are to be created. Furthermore, it should be noted that the ***clk***, ***wave_out***, ***vol***, and ***d_out*** are *not* declared as **ports** in the **module** statement or as global **reg** values. This is not a structural design element but in many ways it is equivalent conceptually to a structural model in that externally declared wires and registers drive and are driven by the *always* block in the manner shown in Figure 3.



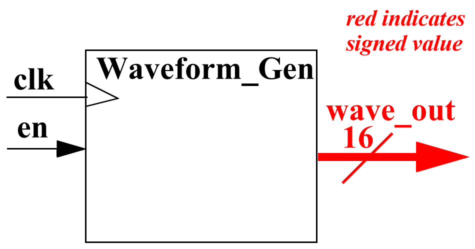
```
always @ (posedge clk)
   begin:DtoA_convert
   // declaring local variable to
   // named section DtoA_convert
   reg signed [3:-28] ge; // error term
   pdm(d_bit,wave_out,{2'b00,vol,6'd0},ge);
   end
```

Figure 3: Cyclic Procedural Implementation of PDM Element that calls the Verilog Task that is named *pdm*

The *DtoA_convert* element is active on the rising edge of the ***clk*** signal. During each of these edges it executes the *pdm* task creating a pulse in accordance with the PDM algorithm. It is driven by two signals. One is the ***vol*** signal which comes as an **input** to the module. The other, ***wave_out***, signal is an internal reg that is driven by the *Waveform_Gen* procedural element.

This waveform generation procedural element is shown in Figure 4. Its purpose is to drive the *wave_out* signal with non-scaled waveform data at a rate that is dictated by an external enabling signal. It does this by referencing a globally defined and pre-initialized lookup table. Figure 4 illustrates the declaration of the *wave_out* signal which is a signed 16 bit signal and the 50 element lookup table that is composed of **signed** 16 bit elements. The ***always*** loop simply responds to a positive edge clock and when this edge occurs it outputs the next element in the lookup table. After the 49th element is accessed the index wraps around making element 0 its next element that will be accessed.

Figure 5 illustrates one method in Verilog of initializing elements such as look-up tables. It utilizes the *initial* procedural section which performs one time operations. In this section the lookup table is initialized using procedural assignment statements for three types of waveforms, square wave, triangular wave, and sine wave. In the template file only one waveform values should be present. The rest are commented out. Instead of listing each assignment statement separately, the square wave initialization utilizes a *for* loop structure which is similar to that encountered in C/C++. It should be noted that this structure does not imply that the operation are synchronized with a clock.



```
reg en;
reg signed [15:0] wave_out;

// 16 bit signed integer 50
// point lookup table
reg signed [1:-14] out_table[0:49];
                            ⋮
always @(posedge clk)
   begin:Waveform_Gen
   // declaring local variable to
   // named section Waveform_Gen
   reg[5:0] index;
   if (en)
      begin
      if (index>=6'd50) index = 6'd0;
      else index = index + 6'd1;
      wave_out = out_table[index];
      end
   end
```

Figure 4: Cyclic Procedural Implementation of Waveform Generating Element

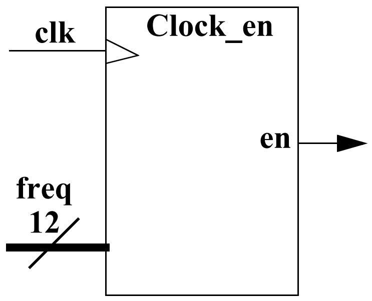
```
reg en;
reg signed [1:-14] wave_out;

  // 16 bit signed integer 50 point
// lookup table for wave function
reg signed [1:-14] out_table[0:49];

initial
  begin:wave_table_init
  // load out_table with
  // 16 bit integer 50 point sine function
  // lookup table
  //                Fixed-Point    Aprox Val
  out_table[0]  = 16'h0000; //  0.000000
  out_table[1]  = 16'h0805; //  0.125333
  out_table[2]  = 16'h0fea; //  0.248690
  out_table[3]  = 16'h178f; //  0.368125
                    ⋮
  out_table[47] = 16'he871; // -0.368125
  out_table[48] = 16'hf016; // -0.248690
  out_table[49] = 16'hf7fb; // -0.125333
  end
```

Figure 5: Using procedural assignment statements in an *Intial* Procedural Section to initialize a signed fixed point 16 bit x 50 element lookup table

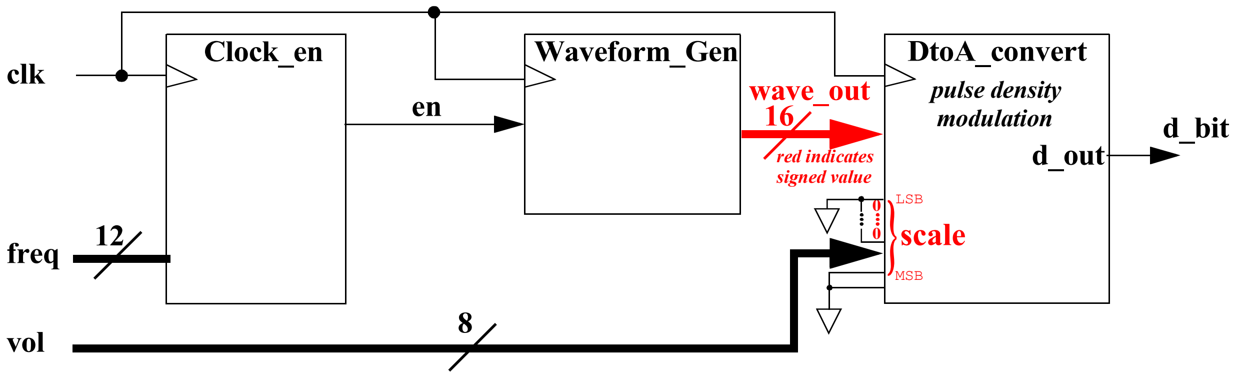
Figure 6 Illustrates the *Clock_en* procedural section that is used to determine the rate at which data is outputted from the *Waveform_Gen* element. The inputs to this module are the system clock and a 12-bit frequency input. The ***clk*** frequency is assumed to be 50 Mhz and the output rate of the enabling pulses are assumed to be 50 times the frequency of the desired waveform to allow all 50 entries of the table to be produced in one cycle. The value of ***freq*** is the desired frequency of the waveform in Hz and in the special case where ***freq*** is set to 0 then the design should stop sending out enabling pulses. This led to the following representation



```
always @(posedge clk)
   begin:Clock_en
      // declaring local variables to
      // named section Clock_en
      reg [19:0] count;
      reg [19:0] base_count;
      if (freq)
         begin
         base_count = 20'd1000000/freq;
         if (count)
            begin
            count = count - 20'd1;
            en=1'b0;
            end
         else
            begin
            en = 1'b1;
            count = base_count;
            end
         end
   end
```

Figure 6: Cyclic Procedural Implementation of Clock Enabling Element used to Determine the Frequency of the Output Waveform

Figure 7 illustrates the overall system representation with the ***clk***, ***freq***, and ***vol*** signals representing the inputs and the ***d_bit*** signal representing the digital output.



```
module lab6(input clk, input [11:0] freq, input [7:0] vol,
    output reg d_bit);
```

Figure 7: Top-level View of Overall Analog Waveform Generator Design

The design can be fully implemented and evaluated on the Altera Teriasic DE2-115 board using the pin assignments shown in Table 1. This assigns the ***clk*** input to the 50 Mhz system clock and the ***vol*** input is assigned to **SW17-SW10**.The 10 msb of the ***freq*** input are assigned to **SW9-SW0** with the 2 least significant bits being assigned to **KEY3** and **KEY2**, respectively.

Table 1: DE2-115 Pin Assignment for base Analog Waveform Generator Design

DE2-115 Connection	Signal Name	FPGA Pin No.	DE2-115 Connection	Signal Name	FPGA Pin No.
50 Mhz DE2-115 Clock	clk	PIN_Y2	SW0	freq[2]	PIN_AB28
Pin D10 UAH ECE Breakout Board	d_bit	PIN_AC19	KEY3	freq[1]	PIN_R24
SW9	freq[11]	PIN_AB25	KEY2	freq[0]	PIN_N21
SW8	freq[10]	PIN_AC25	SW17	vol[7]	PIN_Y23
SW7	freq[9]	PIN_AB26	SW16	vol[6]	PIN_Y24
SW6	freq[8]	PIN_AD26	SW15	vol[5]	PIN_AA22
SW5	freq[7]	PIN_AC26	SW14	vol[4]	PIN_AA23
SW4	freq[6]	PIN_AB27	SW13	vol[3]	PIN_AA24
SW3	freq[5]	PIN_AD27	SW12	vol[2]	PIN_AB23
SW2	freq[4]	PIN_AC27	SW11	vol[1]	PIN_AB24
SW1	freq[3]	PIN_AC28	SW10	vol[0]	PIN_AC24

Assignment

The first part of this laboratory involves the analysis and modification of the existing configuration to observe and measure waveform generation. The second part focuses on expanding the base configuration so that it can be used to generate prerecorded audio. The third part expands this configuration so that it can be used to produce and transmit this audio through a wireless analog AM channel. Students are encouraged to work in groups of two to complete this laboratory, though individual laboratory reports should be written.

Part A: Analog Waveform Generator Evaluation/Simplification

The experimental setup for the base configuration is shown in Figures 8 and 9. The DE2-115 board should be connected to a speaker, oscilloscope, and the digital multimeter in the manner shown. The speaker amplifier should also be turned on. The template design for this portion should be compiled and down-loaded to the DE2-115 board. The digital multimeter should be set to measure volts, V, with a scale range of 2. It is important to note that the oscilloscope should have a ~5K ohm resister in series with the probe as shown in Figure 9. The switches SW17 through SW10 should all be set to the high position and the switches SW9 through SW0 set in a manner that will produce a 1 Khz signal when KEY3 and KEY2 buttons are pressed and held (note: the KEY buttons on the DE2-115 normally produce a Logic 1 and when pressed output a Logic 0).

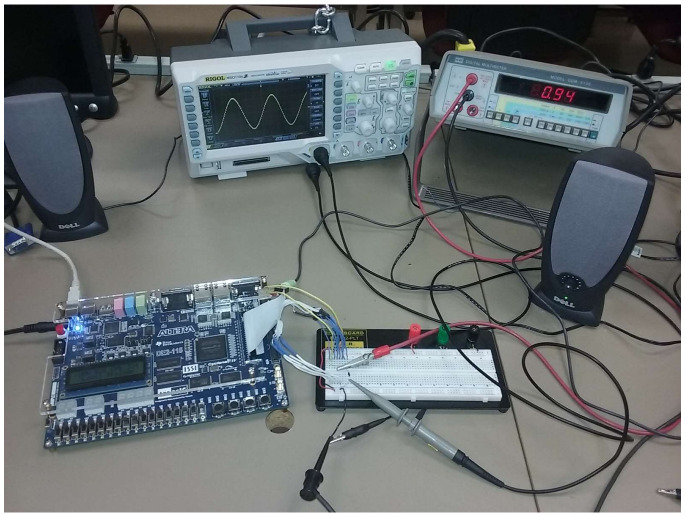


Figure 8: Experimental Setup for Part A of Laboratory -- Overall View

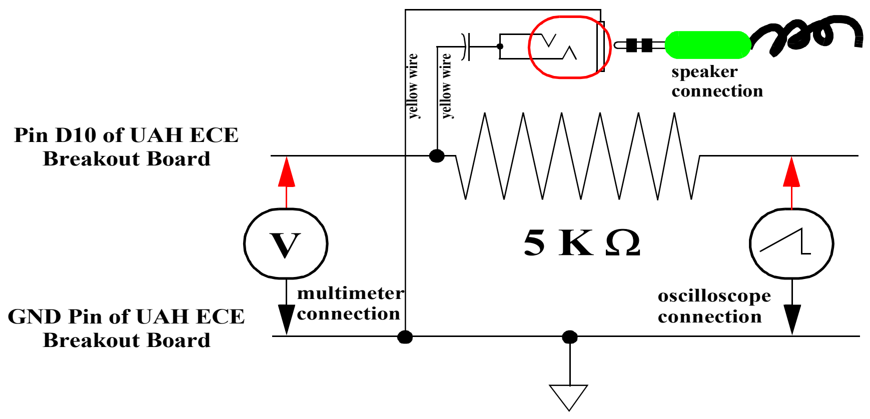


Figure 9: Experimental Setup for Part A of Laboratory -- Detailed View

Uncomment the necessary code sections and synthesize three implementations to separately generate a square wave, triangle wave, and sine wave. Then for each case experimentally answer the following questions and include them as part of your laboratory report.

- What is the effect of varying the switches SW 17 through SW10? off reduces V_{pp} → volume
- What is the effect of varying the switches SW9 through SW0 and KEY3/KEY2? reduces freq → pitch
- What is the slowest Frequency signal that can be produced? 0Hz with all off
- What is the highest Frequency signal? 4.08 KHz
- What is the frequency range that produces audio frequency that can be heard through the speaker? 1Hz to 4.08 kHz
- Set up the switches SW9 through SW0 and KEY3/KEY2 to generate as closely as possible a 1 Khz wave and then complete the following table.

SW 1, 3, 4, 5, 6, 7

1 Khz Waveform -- Square Wave

Oscilloscope Settings

Volts per division 0.86V

Time per division 0.2 ms

Measured (oscilloscope)

Peak-to-Peak Voltage 3.44V

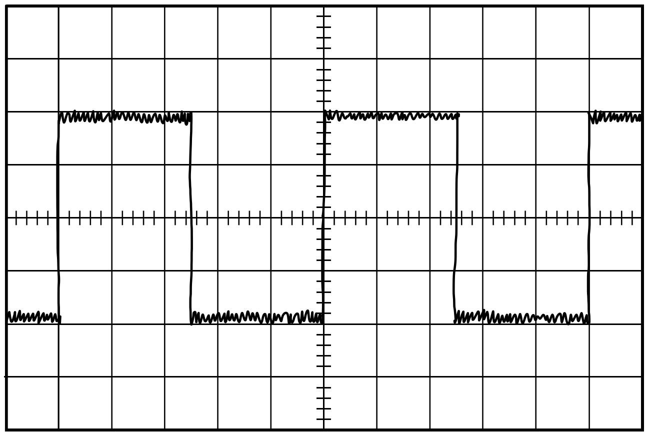
Frequency 1kHz

Measured (Digital Multimeter)

AC Voltage 1.67V

DC Voltage 1.50V

Waveform Sketch



Speaker Observations:

constant, high-pitch

Waveform Observations:

square wave

if possible adjust time mode so that two complete periods of the waveform are displayed

Other Observations:

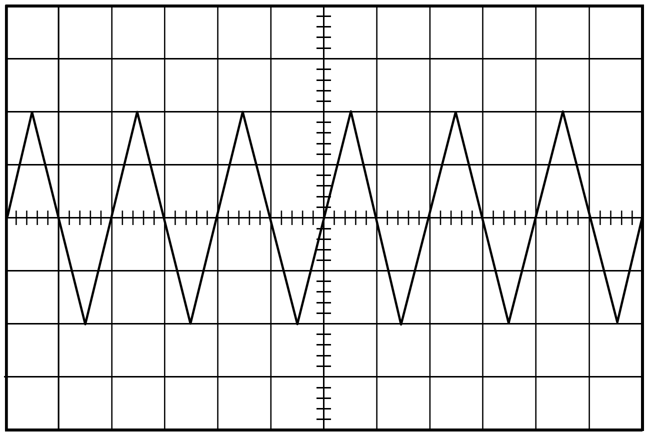
SWO

1 Khz Waveform -- Triangle Wave

Oscilloscope Settings
Volts per division 0.86V
Time per division 0.5 ms
Measured (oscilloscope)
Peak-to-Peak Voltage 3.44V
Frequency 1 kHz
Measured (Digital Multimeter)
AC Voltage 0.96V
DC Voltage 1.66V

Speaker Observations:
same as square

Waveform Sketch



Waveform Observations:
triangle

if possible adjust time mode so that two complete periods of the waveform are displayed

Other Observations:

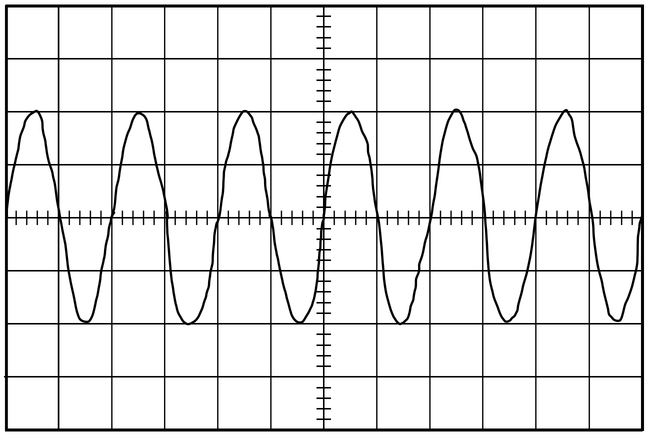
SWO

1 Khz Waveform -- Sine Wave

Oscilloscope Settings
Volts per division 0.88V
Time per division 0.5ms
Measured (oscilloscope)
Peak-to-Peak Voltage 3.52V
Frequency 1kHz
Measured (Digital Multimeter)
AC Voltage 1.17V
DC Voltage 1.62V

Speaker Observations:
same as triangle

Waveform Sketch

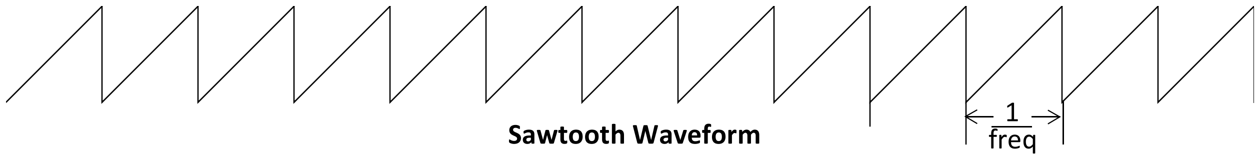


Waveform Observations:
sinusoidal

if possible adjust time mode so that two complete periods of the waveform are displayed

Other Observations:

Modify your design so that it will generate a 1 Khz sawtooth waveform as shown below.



Sawtooth Waveform

1 Khz Waveform -- Sawtooth Wave

Oscilloscope Settings

Volts per division 0.86V

Time per division 0.5ms

Measured (oscilloscope)

Peak-to-Peak Voltage 3.44V

Frequency 1kHz

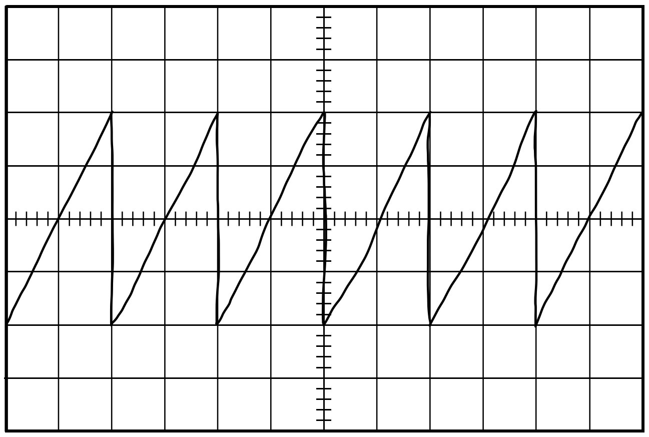
Measured (Digital Multimeter)

AC Voltage 0.96V

DC Voltage 1.63V

Speaker Observations:
high pitch

Waveform Sketch



Waveform Observations:
sawtooth

if possible adjust time mode so that two complete periods of the waveform are displayed

Other Observations:

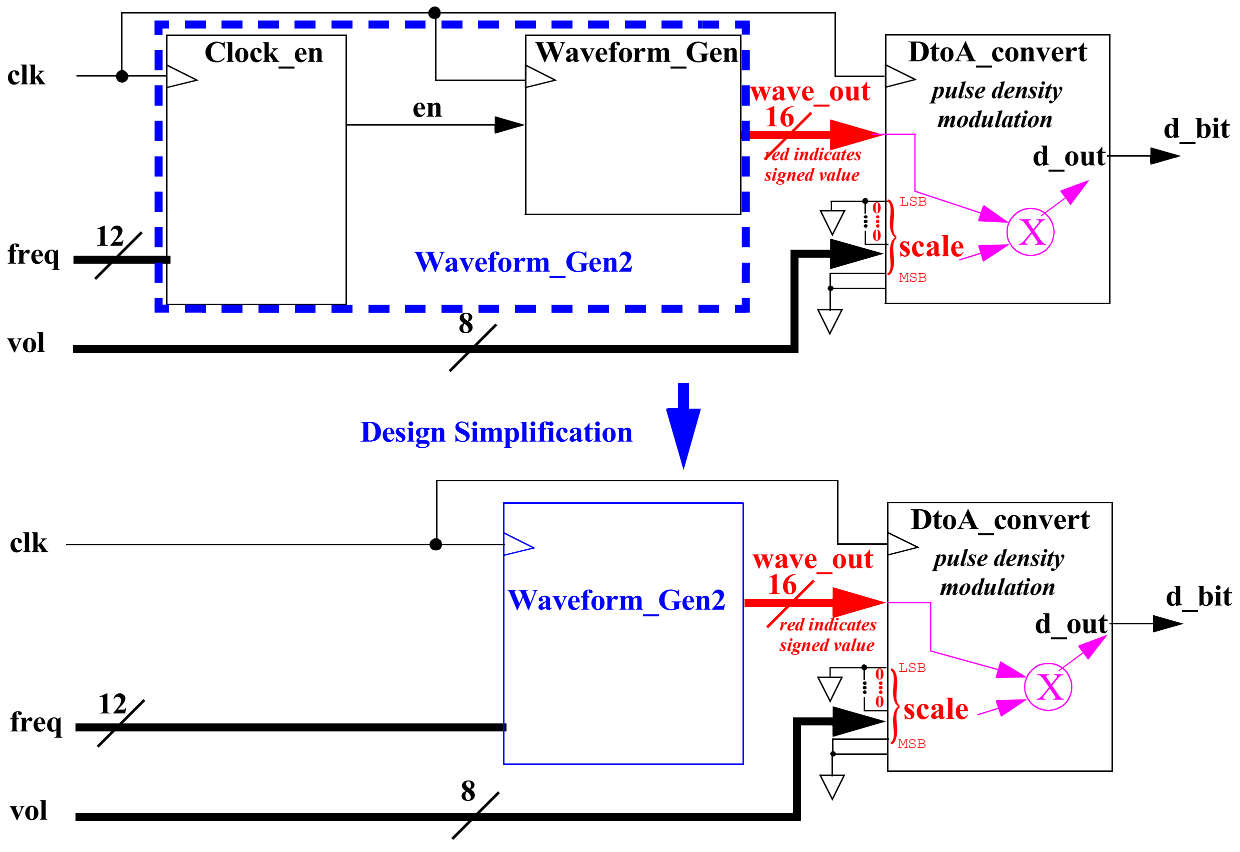
In your report answer the following questions for each of the three waveforms that are generated.

- When the Digital Multimeter is in the AC mode does it measure RMS voltage or something else? What about when it is in the DC mode? If it does not measure RMS voltage then what does it appear to measure?
- How close were the measured frequency to 1 KHz? Explain in your report what could be causing any differences between measured and the frequency that the design is suppose to produce?

Design Simplification

Simplify the design by combining the *Clock_en* and the *Waveform_Gen* procedural blocks into a single procedural block that is named *Waveform_Gen2* as shown in Figure 10. In so doing remove from the design the internal register which is named *en*. This is a higher-level design that is more behavioral in nature than the one that was presented in the template.

- When complete demonstrate the functionality of your the new design to your laboratory instructor before continuing to the next part of the design.



```
module lab6_a (input clk, input [11:0] freq, input [7:0] vol,
   output reg d_bit);
```

Figure 10: Top-level View of Simplified Waveform Generator Design

FPGA $\frac{50 MHz}{8 kHz} = 6250$
max audio

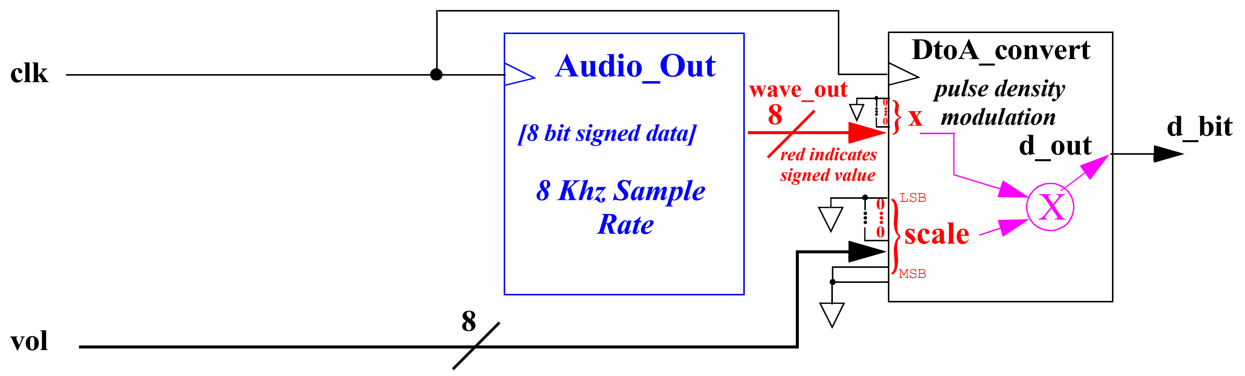
Part B Audio Generator (8 Khz Sample Rate)

In this part of the laboratory modify the *Waveform_Gen2* portion of the design and rename it to *Audio_Out* as shown in Figure 11. Design the *Audio_Out* portion so that it will generate audio from the speaker that is connected to digital output. The speaker should be connected in the same manner as it was in Part A (see Figure 9). Students may use pre-recorded audio that has been sampled at 8000 Hz and has been placed in a text files where they have been expressed in hexadecimal form. This data has been encoded in two's complement form and has been scaled appropriately to allow for full volume to occur when the **scale** input is at maximum value. Employ a two dimensional array and initialize this array in the **initial** section using the **$readmemh** verilog directive. This directive has the following form:

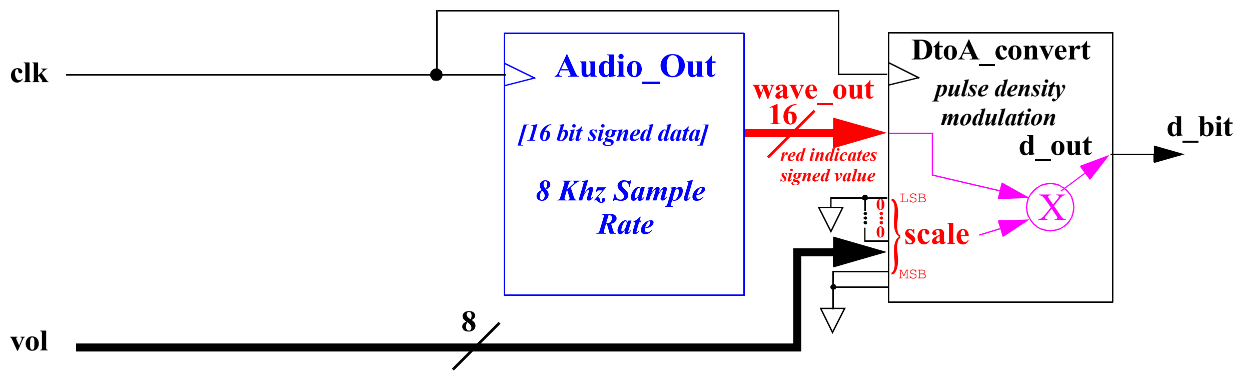
> **$readmemh** ("file_name", 2D_array_name) ;
> where file_name is the name of the file that has numbers in hexadecimal form in sequence from element 0 to the last element in the array with each element placed on a separate line of the file, and 2D_array_name is the name of the two dimensional array to be initialized in the verilog model.

On the Canvas website there are both 8 bit and 16 bit versions of the audio files that can be used. The 8 bit files have the text "*_8_signed*" as part of their name and the 16 bit version of the files have the "*_16_signed*" version as part of their name. They differ from one another by resolution of the recordings. To play back the 8-bit version requires that the lower ordered 8 bits that are fed into the DtoA_convert portion of the circuit be padded with 0's as shown in Figure 11. In the case of the 16 bit version no such padding is needed. The reason for both encodings is to allow for longer audio sessions for the lower resolution to be made.

- Implement at least one version of the audio playback design and demonstrate to your lab instructor the playback of two separate audio files. The design should continuously cycle through the audio clip and replay it when it reaches the end of the clip.
- In your report answer the following question. Given the size of the Altera Cyclone IV E EP4CE115F29C7 FPGA assuming that all the audio memory is stored in embedded RAM and not created from other FPGA logic elements, what is the maximum length (un-compressed) that can be supported for both the 8 bit case and the 16 bit case without exceeding the available resources?



(a) 8 bit signed data version (uses data from "*_8_signed*" named files)



(b) 16 bit signed data version (uses data from "*_16_signed*" named files)

module lab6_b (**input** clk, **input** [7:0] vol, **output reg** d_bit);

Figure 11: Top-level View of Simplified Audio Generator Design

Part C Amplitude Modulation of a Radio Frequency Wave

In the final portion of the laboratory the previous two parts will be combined to allow a sampled 1 Mhz sine wave, which is in the Radio Frequency, RF, range, to be amplitude modulated by 8 Khz sampled audio. This this allows the audio to be broadcast a short distance wirelessly where it can be received by a standard AM receiver. Since the signal voltage levels are very low, and the effective transmission antenna is very small compared the 1 Mhz sine wave the RF emission levels will be well within the range allowed by the Federal Communication Commission, FCC, for unlicensed transmission. Modulation is the process by which a RF signal can be made to transmit useful information. In this case this useful information is the audio. In amplitude modulation, as shown in Figure 12, the amplitude portion of the higher frequency carrier signal is made to change in dynamically in proportion to the modulating signal, *M(t)* which is always non-negative (positive or 0). In the case where the modulation signal is symmetric around a positive and negative V_{max} constant then an offset modulating waveform can be created by adding this V_{max} constant to the two's complement representation. This is what has been done to the waveform files that are available to drive the audio in the design for this part of the laboratory. Once the audio waveform has been offset by this constant then amplitude modulation occurs when this sampled audio signal is multiplied with the sampled higher frequency sine wave as shown in the figure.

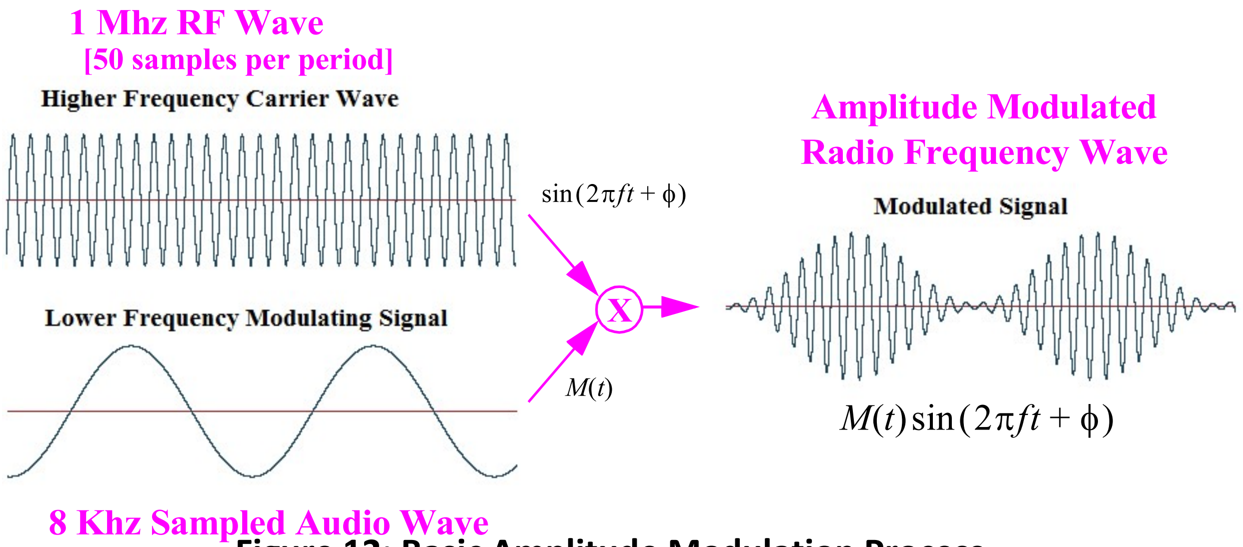
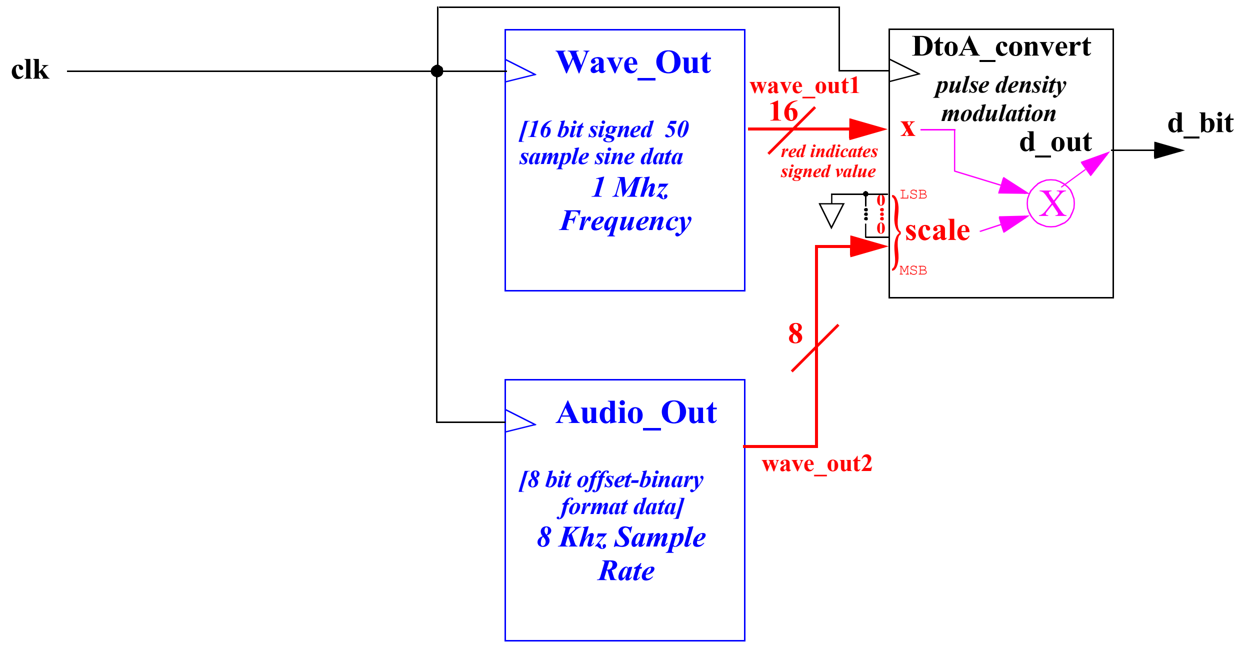

1 Mhz RF Wave
[50 samples per period]
Higher Frequency Carrier Wave
Lower Frequency Modulating Signal
8 Khz Sampled Audio Wave
$\sin(2\pi ft + \phi)$
$M(t)$
X
Amplitude Modulated Radio Frequency Wave
Modulated Signal
$M(t)\sin(2\pi ft + \phi)$

Figure 12: Basic Amplitude Modulation Process

Since the overall product of both waves produce alternating current that is 3 Khz or greater a radio wave is produced and the electrical wiring that connects the speaker will act as an antenna. It is possible to decode the modulation and retrieve the audio signal using a receiver with an AM detector. When the frequency of the carrier is set at 1 Mhz then the signal should detectable using a standard AM receiver that has a frequency range of 0.53 Mhz to 1.70 Mhz.

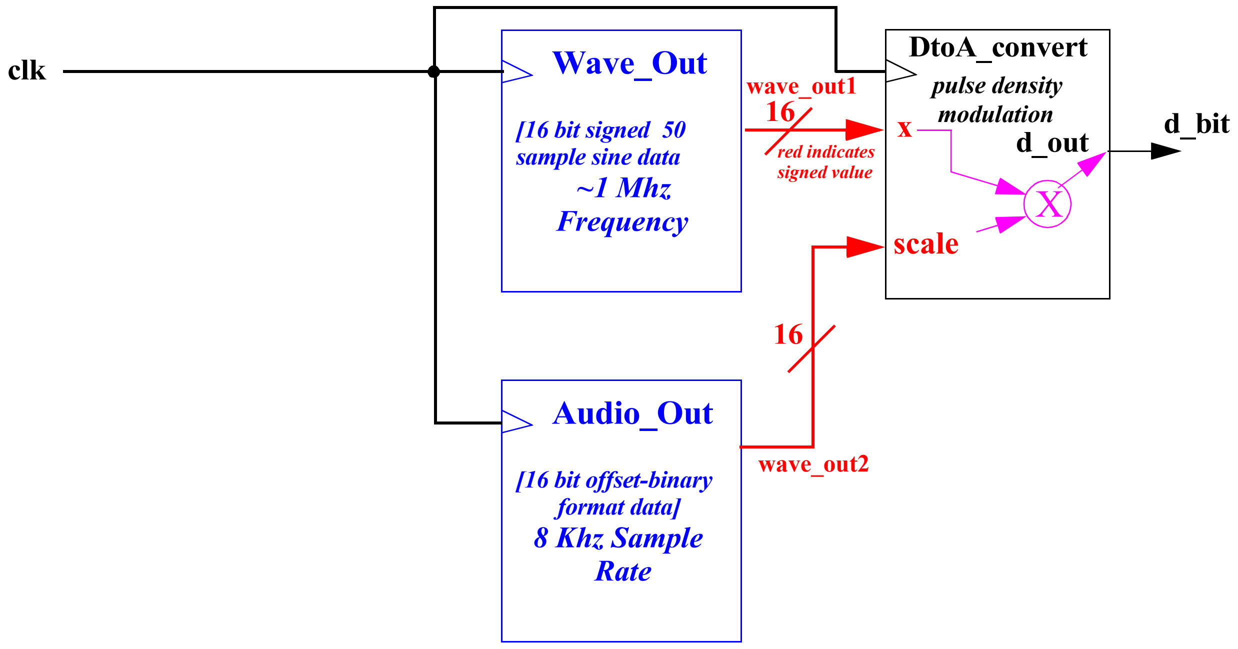
In this part of the laboratory modify the *Audio_Out* portion of the design so that it uses 8 Khz sampled offset data and have it drive the *wave_out2* internal signal that is connected to the *scale* input of the *DtoA_convert* part of the design. If 8 bit data is being used to allow for larger duration audio streams then drive the least significant bits going into the scale input of the *DtoA_convert* portion with logic 0's. If 16 bit data is being used then drive the *wave_out2* reg with the full 16 bit quantity as shown in Figure 13b. In both cases the *Wave_Out* portion of the design should be driven by the values obtained from the corresponding point of the 50 element sine lookup table. This design can be obtained directly from Part A of this laboratory but with all frequency division being removed from the previous module. Even though the speaker is not being used in this portion of the design it should remain connected in the same manner as it was in Parts A and B (see Figure 9). The Speaker amplifier, though, should be turned off.



8 bit offset-binary data version of audio files
(uses data from "*_8_offset* named files)

module lab6_c (**input** clk, **output reg** d_bit);

Figure 13a: Top-level View of AM Transmission Generator Design



16 bit offset-binary data version of audio files
(uses data from "*_16_offset"* named files)

module lab6_c (**input** clk, **output reg** d_bit);

Figure 13b: Top-level View of AM Transmission Generator Design

- Implement at least one version of the AM Transmission Generator design shown in Figure 13 and demonstrate to the lab instructor the RF playback of two separate audio files. The design should continuously cycle through the audio clip and replay it when it reaches the end of the clip. The playback should be received on a standard broadcast band AM radio when it is brought near to the audio connector line of the DE2-115.

Post Laboratory Questions

In addition to the other questions expressed in this lab assignment answer the following three general questions in the final laboratory report.

1. Why was signed arithmetic used in the PDM task? How might signed arithmetic help in terms of modeling the problem?
2. What are the advantages and disadvantages of floating point arithmetic as opposed to integer or fixed-point arithmetic in digital hardware? What favored the use of fixed point in this design?
3. Why are lookup tables often used to generate sine waves? Why not just compute on the fly using a numeric methods to generate the sine function?