CPE 324 Advanced Logic Design Laboratory
Laboratory Assignment #7
Digital to Analog Conversion using Pulse Density Modulation
(10% of Final Grade)

Purpose

The purpose of this laboratory is to expose students to a wider set of Verilog HDL modeling constructs and techniques and to illustrate how these techniques can be used to implement a Digital Signal Processing, DSP, algorithm in hardware. The specific DSP algorithm that is to be modeled in this laboratory is Pulse Density Modulation, PDM, which allows a one-bit digital output pin to mimic the functionality of a true analog output. Students will be asked to verify operation of the core Verilog HDL module by observing the waveform that is produced using an oscilloscope and digital multimeter. They will then modify the original module to playback prerecorded audio through a speaker that is connected to a digital port on the DE2-115 trainer. They will then be asked to reconfigure the module so that it can use pre-recorded audio to amplitude modulate a 1 Mhz sine wave allowing the audio to be received using a standard AM broadcast band receiver.

Verilog Modeling Constructs

This laboratory module utilizes the following aspects of the Verilog HDL language, some of which have not been formally introduced before.
- **signed** data types
- Verilog **task**
- 2-Dimensional Arrays
    declaration
    initialization using individual assignments within an **initial** section
    initialization using the **$readmemh** directive in an **initial** section
- Named procedural blocks
- Local variable declaration and usage
- **`define** directive

Pulse Density Modulation

Pulse density modulation is a common DSP technique that can be employed to create high-fidelity audio output in portable hand held devices. This form of modulation generates a binary bit stream, *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 in a positive logic two-voltage digital system, 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 Altera FPGA of the DE2-115 this corresponds to $V_{max} = 3.3$ V and $V_{min} = 0$ V.) 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 V_{max} to V_{min} range. This voltage will simply reflect the moving average of positive pulses that have been received. This means that the voltage would be V_{max} for bit streams where all *b[i]=1*. It would be V_{min} for cases where all *b[i]=0*. It would be exactly in between V_{max} and V_{min} 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 that produces an output bit of *b[i]=1* or a *b[i]=0* in the bit stream in a manner that incorporates a mathematical representation of the actual digital voltage output and a pending quantization error. This quantization error is computed from the difference from the actual digital output voltage level associated with *b[i]* and the desired analog voltage level. This accumulated error is then negatively fed back in the process loop where it is then either added or subtracted from the cumulative 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 this process can be obtained at http://en.wikipedia.org/wiki/Pulse-density_modulation.

Figure 1 illustrates the basic pulse density modulation algorithm implemented in both C/C++ and in Verilog HDL. In 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 other input is a scale value which will range from 0 to 1. The actual signal to which the PDM method is being applied is the product of the original signal and this scale value times 2 which will be in the range of -1 to 1. 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.

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;

   x=x*scale*2;

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

```
// Pulse Density Modulation Task in Verilog
task pdm(output d_out, input signed [15:0] x,
   input signed [15:0] scale,
   inout signed [31:0] ge);

   // fixed point encoding of 1.0... assuming a
   // signed 3 bit mantisa and a 29 bit fraction
   `define ONE 32'h20000000

   // 32 bit holder of x * scale data
   reg signed [31:0] x_total;
   reg ret_val;

   begin

   x_total = x*scale*2; // scaling value

   if (x_total >= ge)
      begin
      ret_val = 1;
      ge = ge + (`ONE - x_total);
      end
   else
      begin
      ret_val = 0;
      ge = ge - (`ONE + x_total);
      end
   d_out = ret_val;
   end
endtask
```
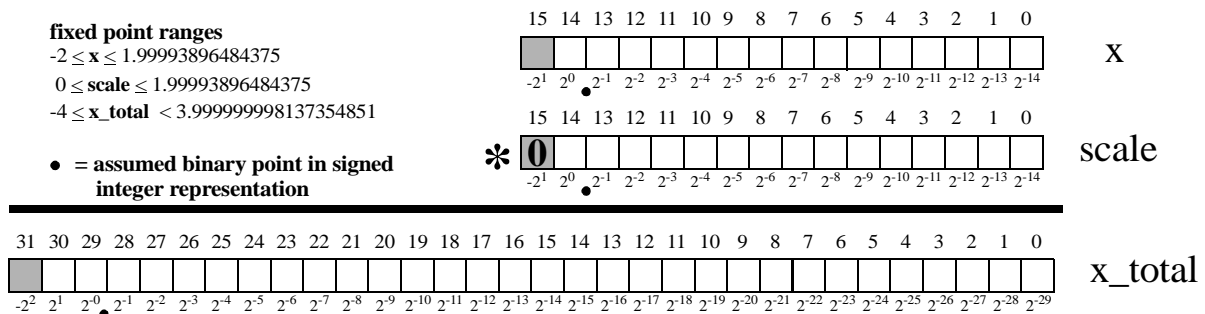
Figure 1: Basic Pulse Width Modulation Algorithm
(C/C++ floating point and Verilog HDL Fixed Point Representations)

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*2* should always be between -1 and 1, the alpha sigma method employees 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 to create such circuitry is much less that for a given floating point representation. The speed of fixed point operations is also significantly faster and much more predictable.

fixed point ranges
$-2 \leq x \leq 1.99993896484375$
$0 \leq scale \leq 1.99993896484375$
$-4 \leq x\_total < 3.999999998137354851$

● = assumed binary point in signed integer representation

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | X |
| -2^1 | 2^0 | 2^{-1} | 2^{-2} | 2^{-3} | 2^{-4} | 2^{-5} | 2^{-6} | 2^{-7} | 2^{-8} | 2^{-9} | 2^{-10} | 2^{-11} | 2^{-12} | 2^{-13} | 2^{-14} |

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		scale
0																
-2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}

* (multiply)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | x_total |
| -2^2 | 2^1 | 2^0 | 2^{-1} | 2^{-2} | 2^{-3} | 2^{-4} | 2^{-5} | 2^{-6} | 2^{-7} | 2^{-8} | 2^{-9} | 2^{-10} | 2^{-11} | 2^{-12} | 2^{-13} | 2^{-14} | 2^{-15} | 2^{-16} | 2^{-17} | 2^{-18} | 2^{-19} | 2^{-20} | 2^{-21} | 2^{-22} | 2^{-23} | 2^{-24} | 2^{-25} | 2^{-26} | 2^{-27} | 2^{-28} | 2^{-29} |

Verilog pdm Task
```
x_total = x*scale*2; // scaling value
```
Figure 2: Signed Fixed-Point Range Analysis internal to the

There is no fixed-point type in Verilog but fixed-point can easily be emulated using signed integers. Fixed-point assumes that there are a fixed 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. As in many cases with low-level number representation, the value of the number depends upon the rules one uses for its interpretation. As long as we maintain the sign bit it is possible to assume a different weighting of the numbers that represent the integers by assuming that the binary point is associated with any bit position. This is shown for the number **x** where bit positions 13 through 0 are considered to be to the right of the binary point and bit positions 14 and 15 are to the left of this point. Because the **signed** type is used bit 15 will represent the sign bit. This assumed weighting makes it possible to approximate numbers in the range of -2 to 2 as shown in the figure. Since Verilog is unaware of this alternative interpretation we are imposing, to obtain the actual number one would divide the integer number by the factor of 2 raised to the number of bit positions to the right of the binary point (or shift right that many bit positions). Thus the number 1.0 for input *x* in Figure 2 could be represented in Verilog as 16'h4000, or as 16'd16384. (Using the same rationale the number 1.0 for *x_total* in Figure 2 is encoded as 32'h200000 in the verilog representation of the **pdm** task in Figure 2).

The *scale* variable is a 16 bit **signed** variable. This variable is created by concatenating a 0 to a 15 bit unsigned quantity in the call to the Verilog **task**. This allows the *x_total=x*scale*2* multiplication to occur in a signed manner where the scale value always being interpreted as a positive value. The multiplication of two 16 bit numbers will fit within a 32 bit space. The results are mathematically correct if we interpret the **x_total** variable as bits 31, 30, and 29 representing the integer portion and the remaining 29 bits representing the fractional component. This gives a range from -4 to just less than 4 which is more than enough to accommodate are desired range. Figure 3 illustrates the case where only 8 bits of the scale variable are to be used in which case the lower order 7 bits are assumed to be zero. This was done to facilitate the DE-2 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 [31:0] ge; // error term
   pdm(d_bit,wave_out,{1'b0,vol,7'd0},ge);
   end
```
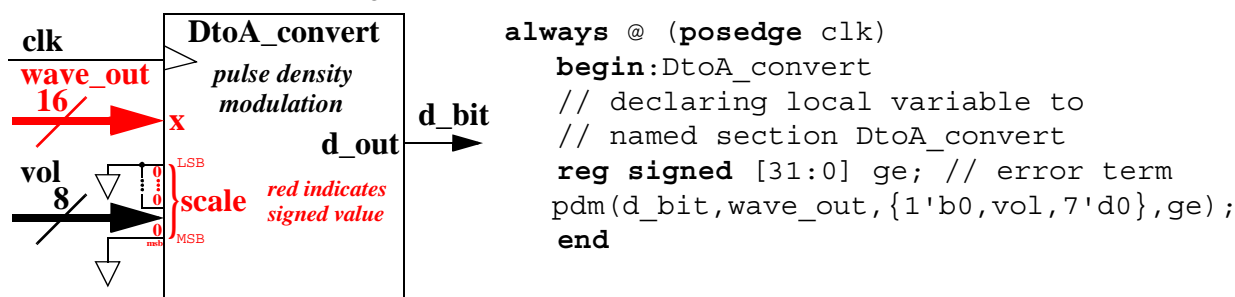
Figure 3: Cyclic Procedural Implementation of PDM Element that calls the Verilog Task that is named *pdm*

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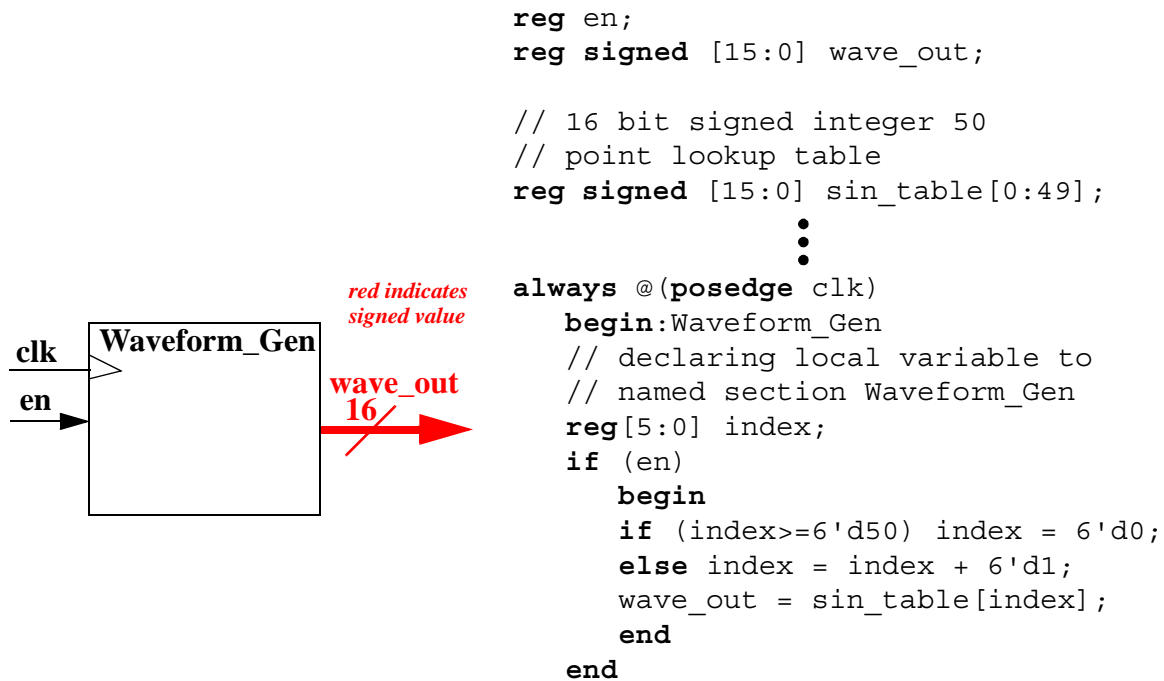
```verilog
reg en;
reg signed [15:0] wave_out;

// 16 bit signed integer 50
// point lookup table
reg signed [15:0] sin_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 = sin_table[index];
      end
   end
```

red indicates signed value

```
clk ──▷ Waveform_Gen
en ──▶
        wave_out
        16 ══▶
```

Figure 4: Cyclic Procedural Implementation of Waveform Generating Element

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

// 16 bit signed integer 50 point lookup table
reg signed [15:0] sin_table[0:49];

initial
  begin:Out_table_init
  // 16 bit integer 50 point sine function
  // lookup table
  sin_table[0] = 16'd0;
  sin_table[1] = 16'd1026;
  sin_table[2] = 16'd2037;
  sin_table[3] = 16'd3015;

                  .
                  .
                  .

  sin_table[47] = -16'd3015;
  sin_table[48] = -16'd2037;
  sin_table[49] = -16'd1026;
  end
```

Figure 5: Using procedural assignment statements in an *Intial* Procedural Section to Initialize a signed lookup 16 bit x 100 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** was meant to represent the desired frequency of the waveform in Hz and in the case where **freq** was set to 0 then no further enabling pulses should be sent. 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
```
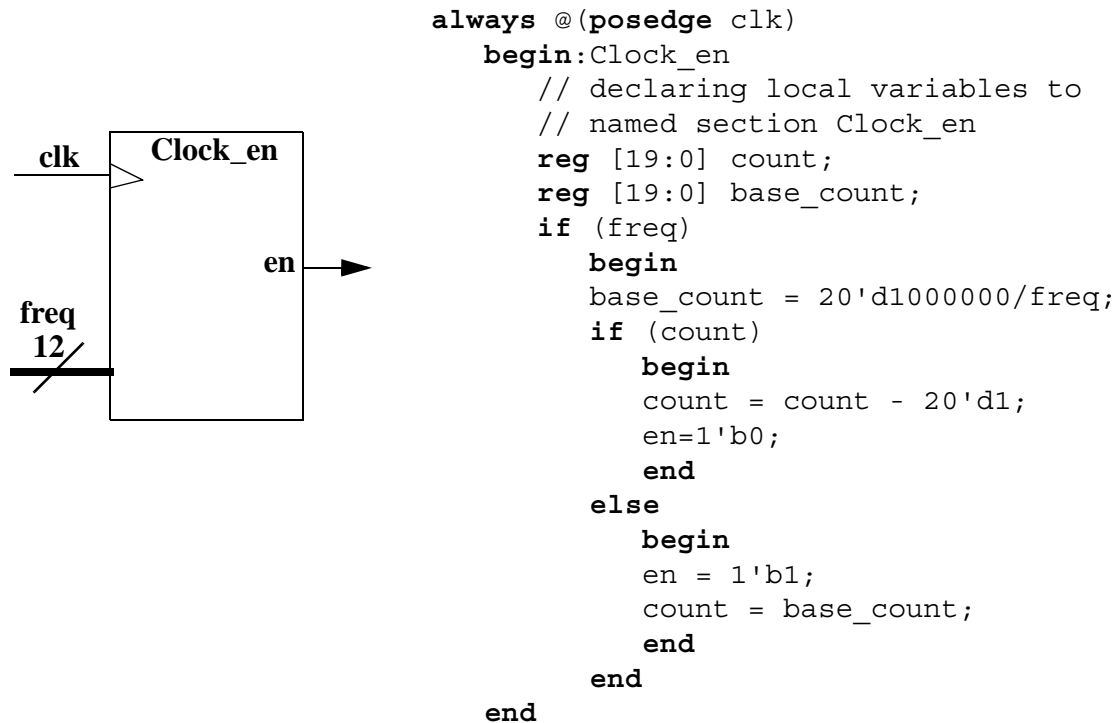
Figure 6: Cyclic Procedural Implementation of Clock Enabling Element used to Determine the Frequency of the Output Waveform

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 lab7 (input clk, input [11:0] freq, input [7:0] vol,
    output reg d_bit);
```
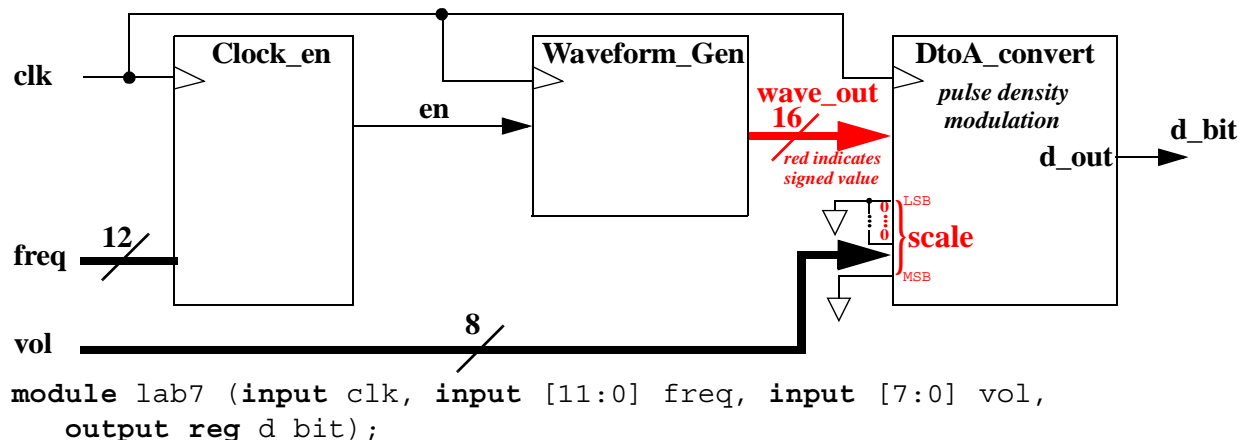
Figure 7: Top-level View of Overall Analog Waveform Generator Design

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
blue/green0 expansion	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 and the second two parts allow students to expand upon the base configuration to create new configurations that can be used to produce audio both directly and through standard analog wireless techniques. Students are encouraged to work in groups of <u>two</u> to complete parts of 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 downloaded to the DE2-115 board.The digital multimeter should be set to measure volts, V, with a scale range of 20. 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 engaged (not the KEY buttons on the DE2-115 normally produce a Logic 1 and when pressed output a Logic 0).
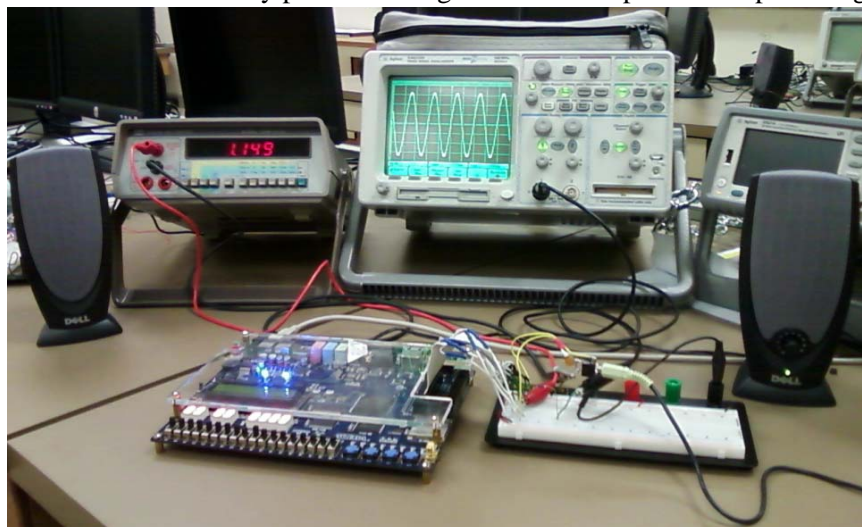

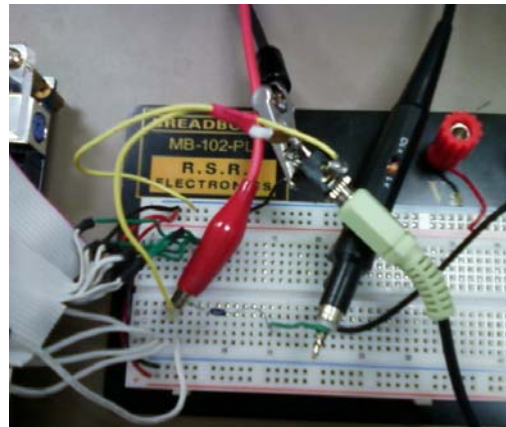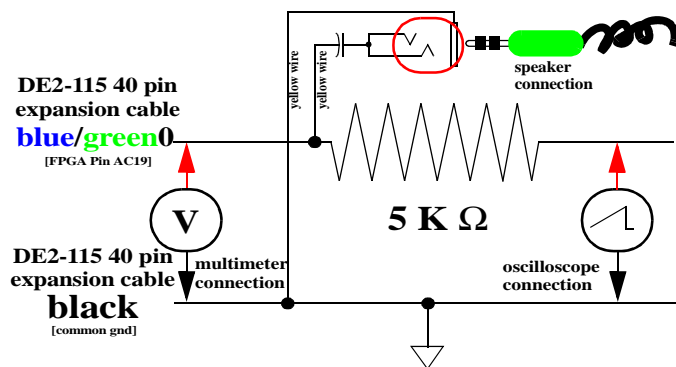**Figure 8: Experimental Setup for Part A of Laboratory -- Overall View**

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

Then 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?
- What is the effect of varying the switches SW9 through SW0 and KEY3/KEY2?
- What is the slowest Frequency signal that can be produced?
- What is the highest Frequency signal?
- What is the frequency range that produces audio frequency that can be heard through the speaker?
- 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.

1 Khz Waveform -- Maximum Amplitude

Oscilloscope Settings
    Volts per division _____
    Time per division _____

Measured (oscilloscope)
    Peak-to-Peak Voltage _____
    Frequency _____

Measured (Digital Multimeter)
    AC Voltage _____
    DC Voltage _____

Speaker Observations:
_____
_____
_____
_____

Waveform Observations:
_____
_____
_____

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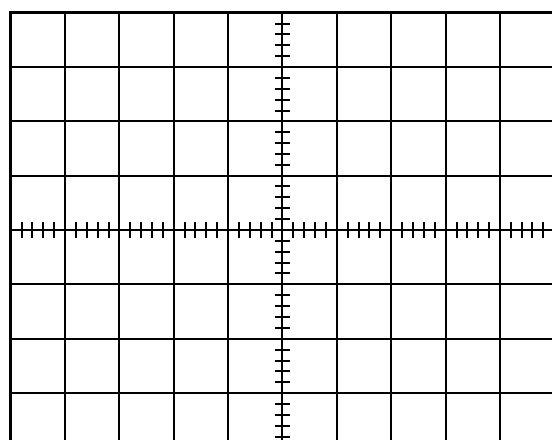
if possible adjust time mode so that two complete periods of the waveform are displayed

Other Observations:
_____
_____

- What is the characteristic of the waveform that is produced? (i.e. what type of wave is produced)?
- For an AC sinusoidal waveform the effective DC voltage (RMS Voltage) is equal to $0.5 V_{\text{Peak-to-Peak}} / (\sqrt{2})$. 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.
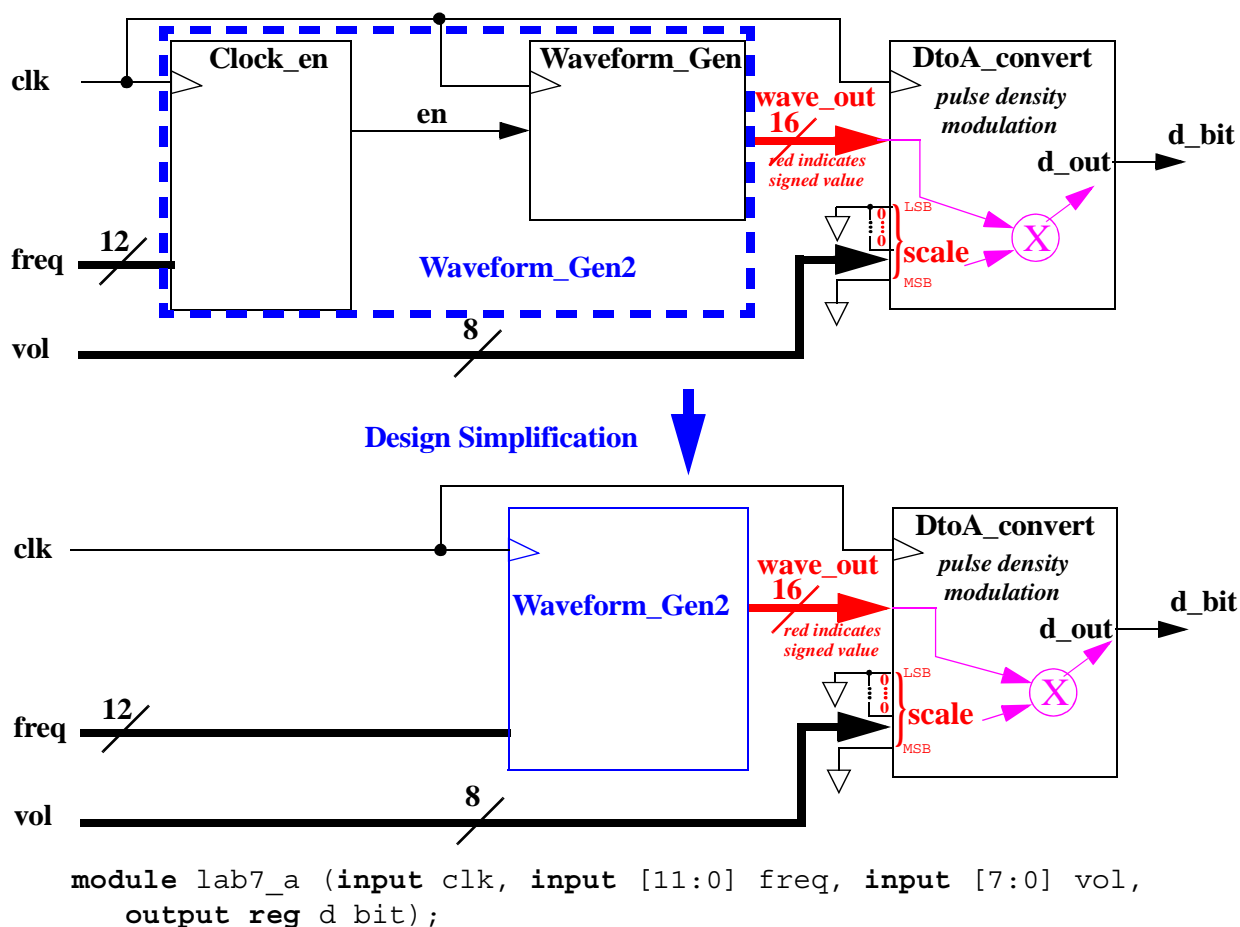


```
module lab7_a (input clk, input [11:0] freq, input [7:0] vol,
    output reg d_bit);
```

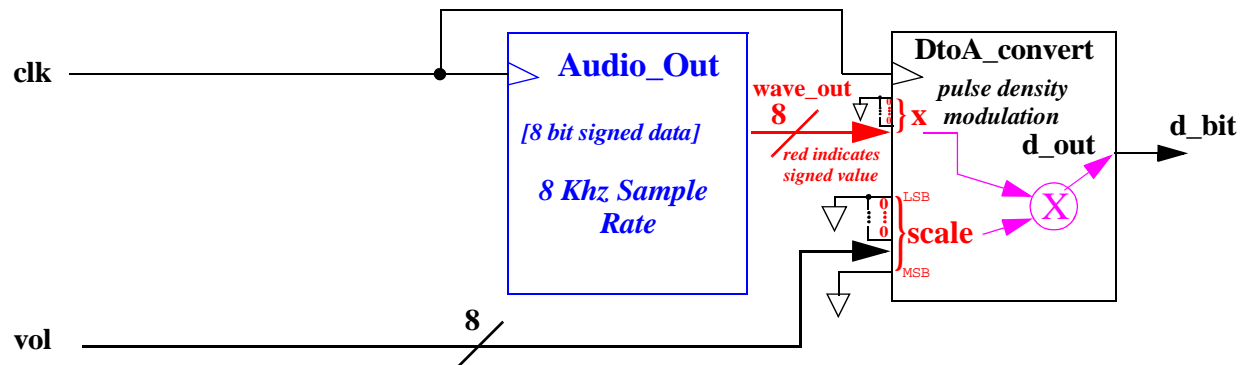**Figure 10: Top-level View of Simplified Waveform Generator Design**

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.

(a) 8 bit signed data version (uses data from "*_8_signed*" named files)
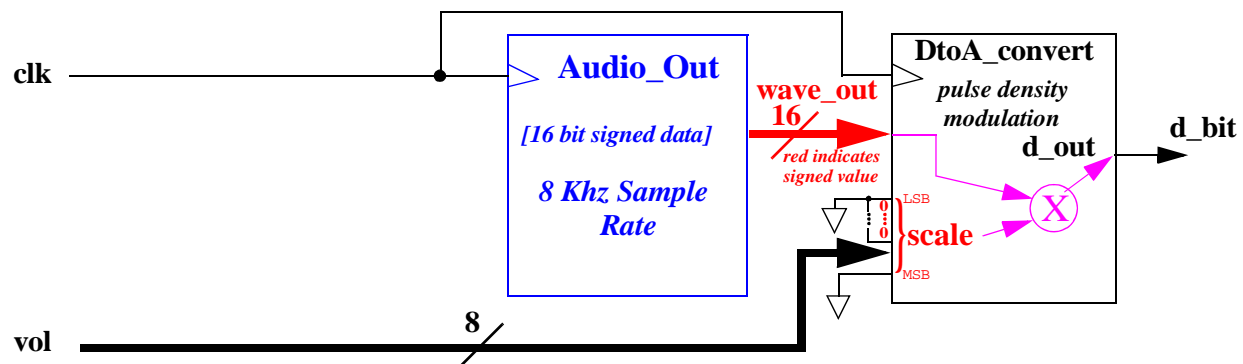
(b) 16 bit signed data version (uses data from "*_16_signed*" named files)

```
module lab7_b (input clk,  input [7:0] vol, output reg d_bit);
```

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

- 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?

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.
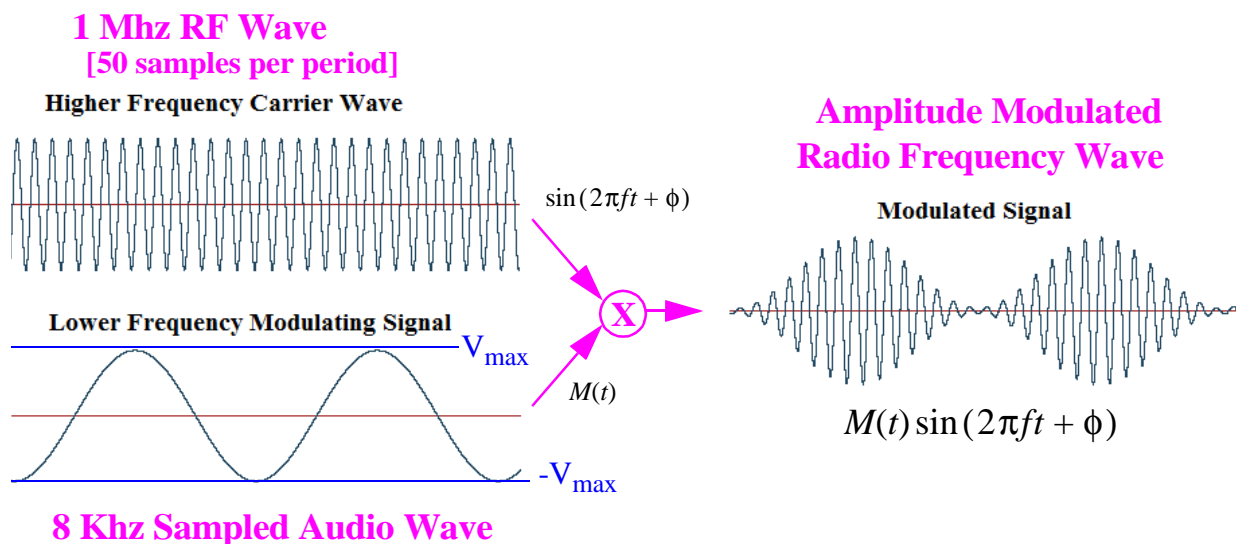

Figure 12: Basic Amplitude Modulation Process

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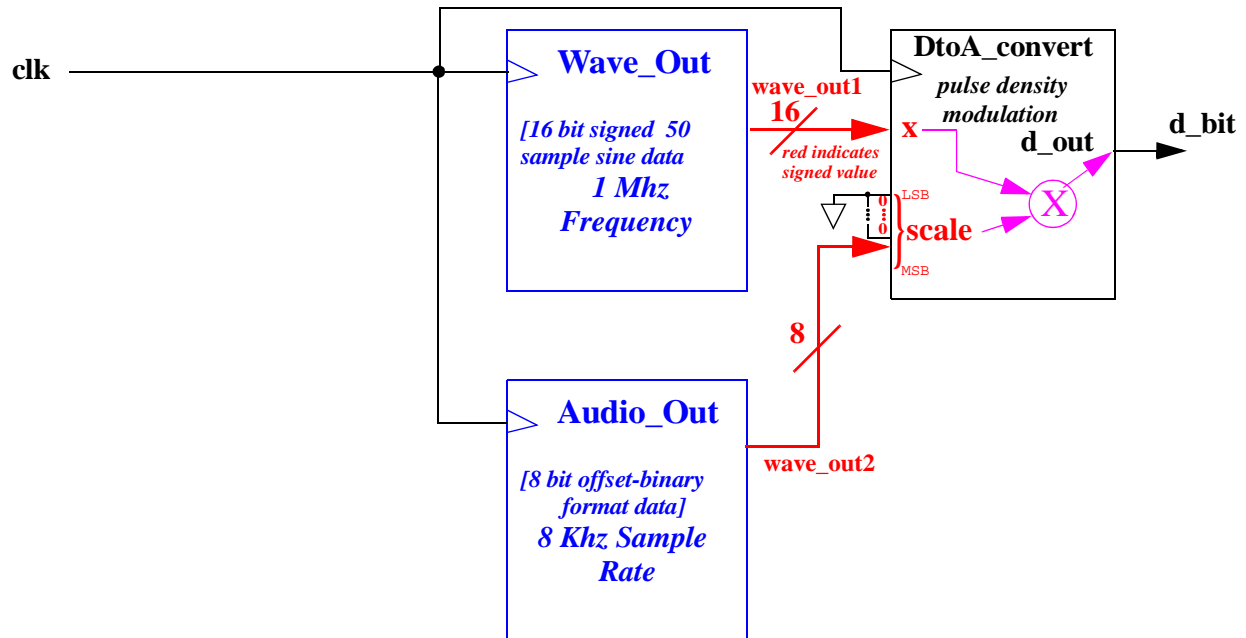
8 bit offset-binary data version of audio files
(uses data from "*_8_offset named files)**

module lab7_c (**input** clk, **output reg** d_bit);

Figure 13a: Top-level View of AM Transmission Generator Design
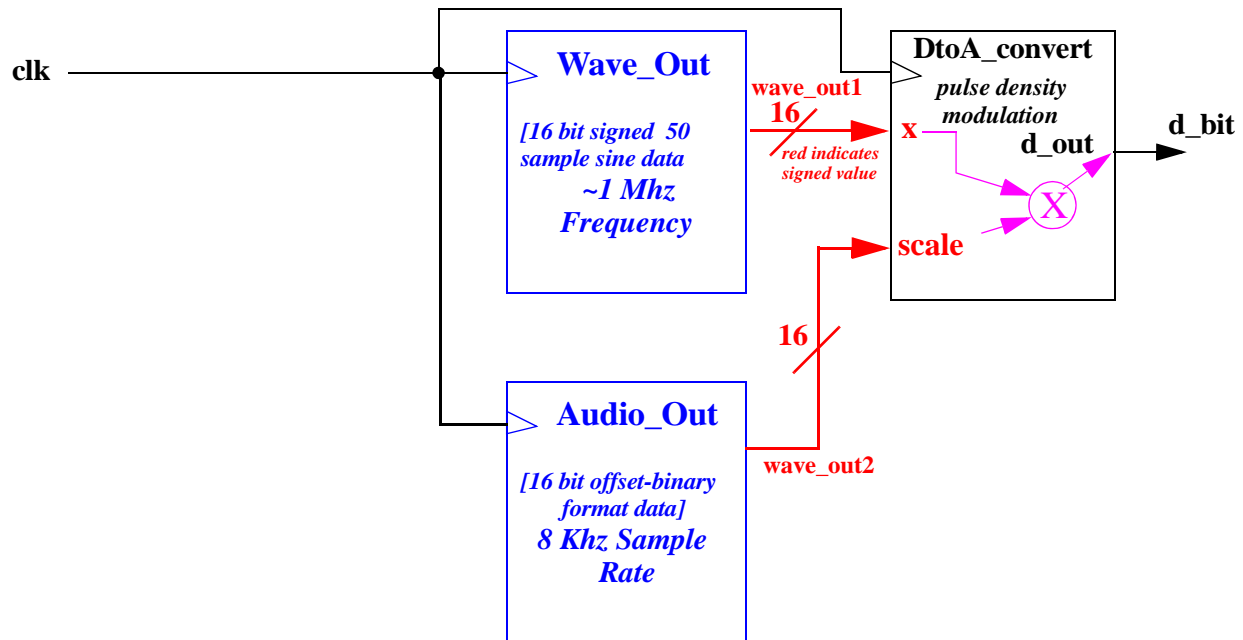
**16 bit offset-binary data version of audio files
(uses data from "*_16_offset" named files)**

```
module lab7_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? 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?