ECSE – 436 Lab 2
Group 1

Muhammad Taha - 260505597
Chrouk Kasem 260512917

Question 1

a) Refer to Figure 18 in appendix for the block diagram of rate-half encoder. The “hello world” block is the bit stream generator as seen in Figure 19 in appendix, which was implemented using the provided code from Verilog documents. The results (c1 and c2) from the rate-half encoder, obtained via signal-tap analysis is shown in Figure 1 below.

Figure 1: SignalTap Analysis of rate-half encoder

b) Refer to Figure 20 in appendix for the filter_demo block diagram, which implements an echo system. The “2048_32bit_regs” uses the provided 32-bit register Verilog file to create a delay for the input sound. The 32-bit register was cascaded for a total of 2048 times to create a noticeable delay. Adding this delayed sound to the original creates the echo.

Question 2

a1) Refer to Figure 23 in Appendix to find the script for the decoder of (8,4,4) Hamming code with comments explaining it. The script finds the shortest distance because each codeword in the codebook and received signal. The distance between one codeword and the received vector is found by:

$$\sum_{i=1}^{8}(ri - ci)^2$$

This distance is calculated for all sixteen codewords in (8,4,4) Hamming code and codeword with minimum distance is the one matched.

Using the decoder with given received vector = [0.54, -0.12, 1.32, 0.41, 0.63, 1.25, 0.37, -0.02]. The minimum distance was matched to the following codeword:

0 0 1 1 1 1 0 0

a2) The number of floating operations can be found by analysing the additions, multiplications, subtractions to find codeword. Knowing that we find distance between one codeword and received vector using:

$$\sum_{i=1}^{8}(ri - ci)^2$$

This includes 8 subtractions, 8 multiplications and 7 additions which gives a total 23 flops to compute distance between one codeword and received vector. This is done for all 16 codewords, which results in number of flops:

16 × 23 = 368 flops

b1) The trellis for the (8,4,4) Hamming code with 4 states at most per stage is shown in Figure 2:

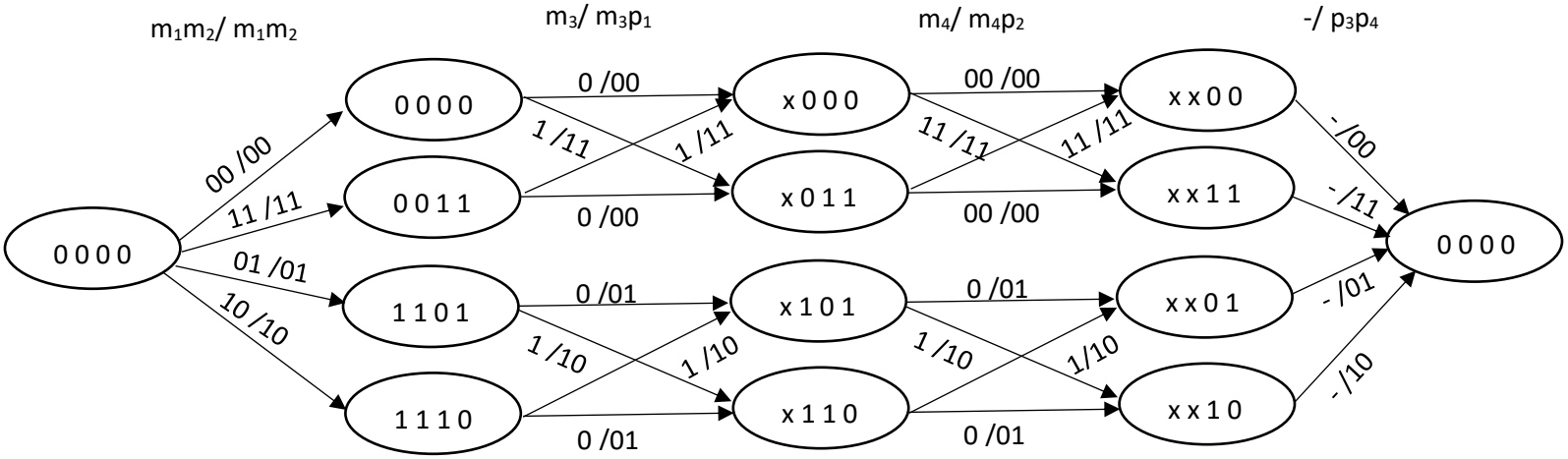


Figure 2: (8,4,4) Hamming code trellis

b2) The appropriate edge weights for the individual trellis edges is calculated using this equation below:

$$d(e) = |r_i - c_i(e)|^2 + |r_{i+1} - c_{i+1}(e)|^2$$

With received vector = [0.54, -0.12, 1.32, 0.41, 0.63, 1.25, 0.37, -0.02], the edge weights calculated using previous equation are shown in trellis in Figure 3.

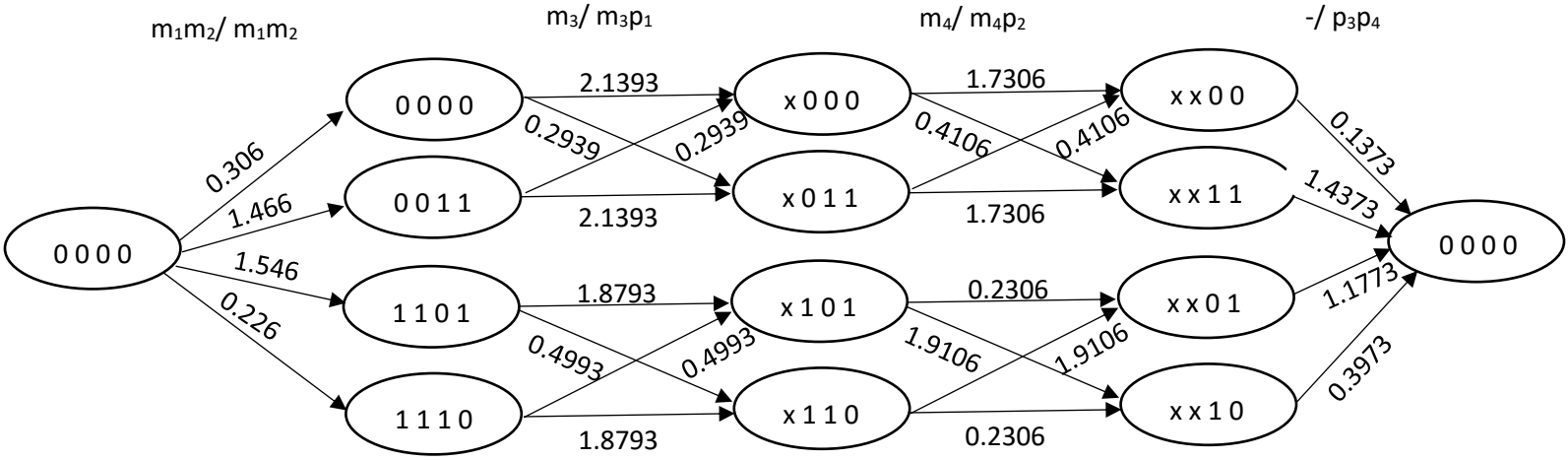
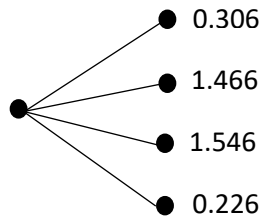


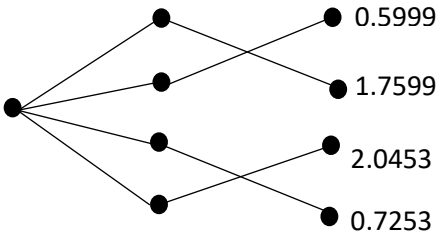
Figure 3: Optimized (8,4,4,) trellis using Viterbi algorithm

This shows the edges at each stage, then the survivors of each stage will be searched, by ADD-COMPARE-STORE step.

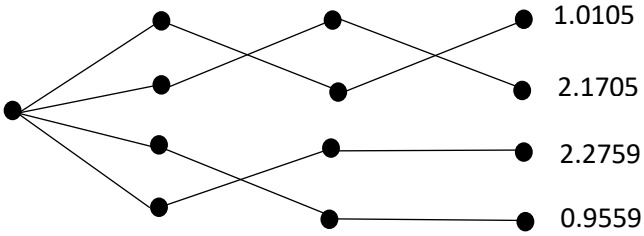
The survivors of stage one are:



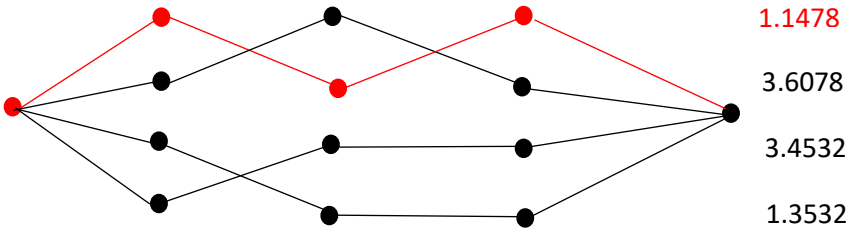
The survivors of stage two are:



The survivors of stage 3 are:



The survivor paths of stage 4 and the least distance path is shown in red:



This path is the one corresponding to the 0 0 1 1 1 1 0 0 codeword.

b3) The total number of flops using Viterbi decoder is reduced to 23 flops. This is achieved by symmetry of the trellis. The equation used to calculate the edge distances, can be expanded as:

$$d(e) = |r_i - c_i(e)|^2 + |r_{i+1} - c_{i+1}(e)|^2 = r_i^2 + r_{i+1}^2 \pm 2r_i \pm 2r_{i+1} + 2$$

At each stage, r_{i+1}^2, r_i^2, and 2 are common among all paths. This reduces calculating the edge distances to:

$$d(e) = \pm 2r_i \pm 2r_{i+1}$$

Taking out the common factor of 2, reduces the calculation further to:

$$d(e) = \pm r_i \pm r_{i+1}$$

The edge distances at stage one will be $r_1 + r_2$, $-r_1 - r_2$, $-r_1 + r_2$, and $r_1 - r_2$. The negation of $r_1 + r_2$ gives $-r_1 - r_2 = -(r_1 + r_2)$, and negation of $-r_1 + r_2$ gives $r_1 - r_2 = -(-r_1 + r_2)$. Negation is not considered a floating point operation in two's complement representation, sign bit is changed to negate result. This reduces calculating the edge distance at stage one to 2 flops. This symmetry property for calculating edge distances extends to stages 2, 3, and 4 giving a total of 8 flops for this step.

For the second step of ADD-COMPARE-STORE, the number of flops for this stage is 15 flops. This can be achieved using inequalities to determine what choice to make for survivor paths. For the first stage, all paths are taken. At the second and third stages, taking a and b as real positive integers that represent edge distances calculated at r_i and r_{i+1}, there are four options for each state:

Option 1: a>0 and b>0 then winner path is –(a+b)

Option 2: a<0 and b<0 then winner path is a+b

Option 3: a>0 and b<0 then winner path is decided by taking calculation which takes one flop

Option 4: a<0 and b>0 then winner path is decided by taking calculation which takes one flop

The number of flops at most for inequality calculation at each state in stages 2 and 3 is 1 flop. For all four states, the most number of flops to find partial survivors at stages 2 and 3 is:

$$4 \text{ flops per stage} \times 2 \text{ stages} = 8 \text{ flops}$$

With 4 more flops to add distance of survivor paths and 3 flops to compare end survivor paths and choose codeword, this makes total number of flops for this step: 8 + 4 + 3 =15 flops.

The total number of flops for both steps in Viterbi decoder is: 15 + 8 = 23 flops.

c1) The MATLAB code for Viterbi decoder for (8,4,4) Hamming Code is shown in Figures 21 and 22 in Appendix. The code was into two steps: labelling edge distance, and ADD-COMPARE-STORE.

c2) Power of received signal is 1, making the SNR = $1/\sigma^2$. The Gaussian noise at each SNR level is calculated by multiplying square root of the variance, standard deviation by randn(1, 8). Variance is calculated using:

$$\sigma^2 = 10^{-(\frac{SNR\ in\ dB}{10})}$$

The noise will be equal to:

$$\text{Noise} = \sigma * \text{randn}(1,8)$$

This noise is added to each codeword, then passed to Viterbi decoder to retrieve original codeword. The number of bit errors in message bits is taken to give BER over 10,000 packets or when 100 codeword errors are reached.

The BER vs. SNR plot for Viterbi decoder performance with additive Gaussian noise is shown in Figure 4**.**

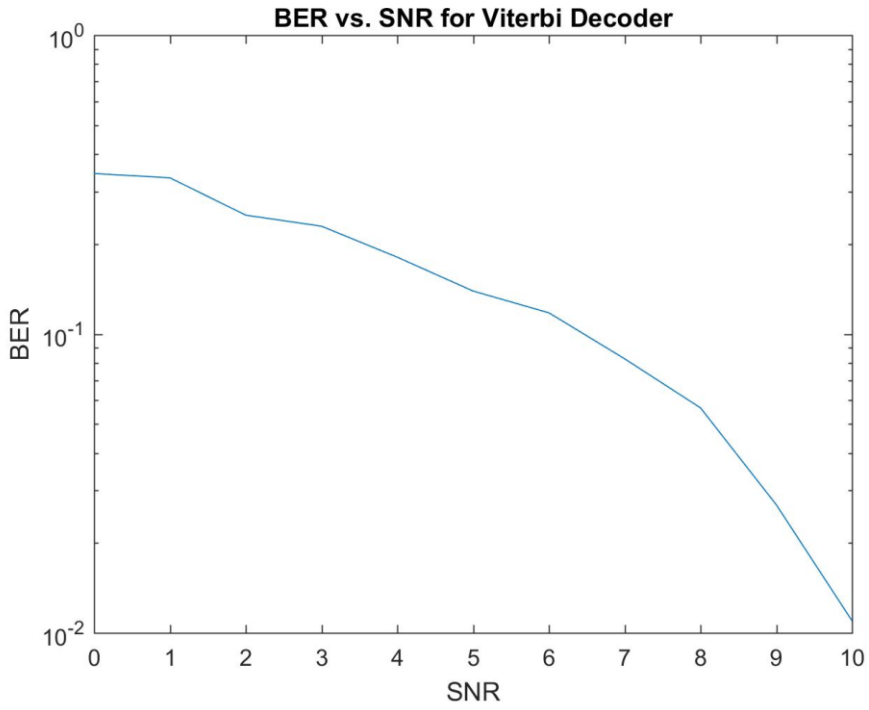


Figure 4: BER vs SNR plot for Viterbi

Question 3

a) The plot in Figure 5 below shows the discrete plot of impulse response (h[n]).

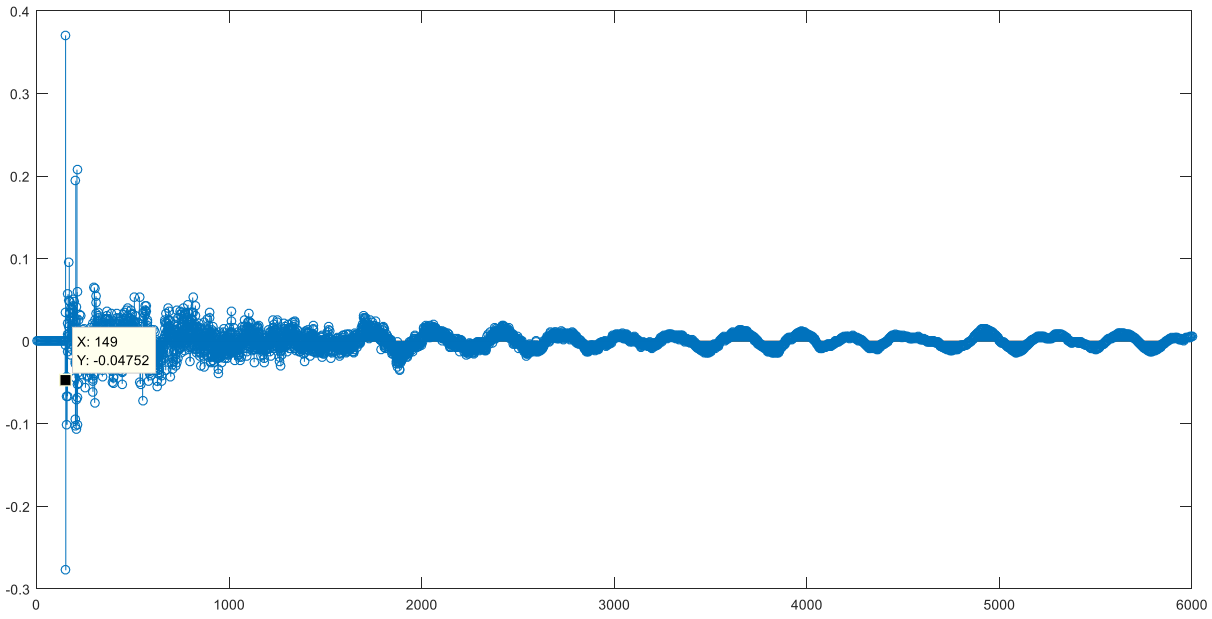


Figure 5: Impulse response plot

The frequency of the audio clip and the impulse is 16000 Hz, so the delay = 149/16000 = 0.0093 seconds. Speed of sound = 340 m/s, so distance between microphone and the person is d = 340*0.0093 = 3.1662 m.

This result was confirmed with MATLAB as well, by looping over the impulse vector and finding the index where the first non-zero value occurs.

b) The sound is still audible after convolution, although there some noise could be also be heard. The background noise seems to have degraded the sound quality of the original audio. The code for this convolution can be seen in Figure 24 in appendix.

c) An delayed version of the sound can be heard along with the original, therefore creating the impression of an echo. The code to create this echo sound is in Figure 25 in appendix.

d) The sound is played backwards by reversing the original sound vector. This is achieved by using the `flipud` function in MATLAB. Playing the backwards signal resulted in speech which could not be understood. The code for this reversal and playback can be seen in Figure 26 in appendix.

e) Table 1 below shows the results observed after playing the audio at different frequencies.

Table 1: Observations of playing the sound at different frequencies

Frequency	Comments
13000	The sound seems slower and the voice is now thick, low-pitched voice
14500	The sound is quite like the original but still slow, quite thick, low-pitched voice
17000	The sound is quite similar but is a faster, higher pitched version of the original
18500	The sound is much faster and higher pitched version of the original, and seems much higher than the previous frequency
20000	At this frequency, the sound is way high pitched and much faster than the original

f) The plot in Figure 6 shows the Fast Fourier Transform (FFT) plot, comparing the original signal, with the sub-sampling signals. The MATLAB code for aliasing can be seen in Figure 27 in appendix.

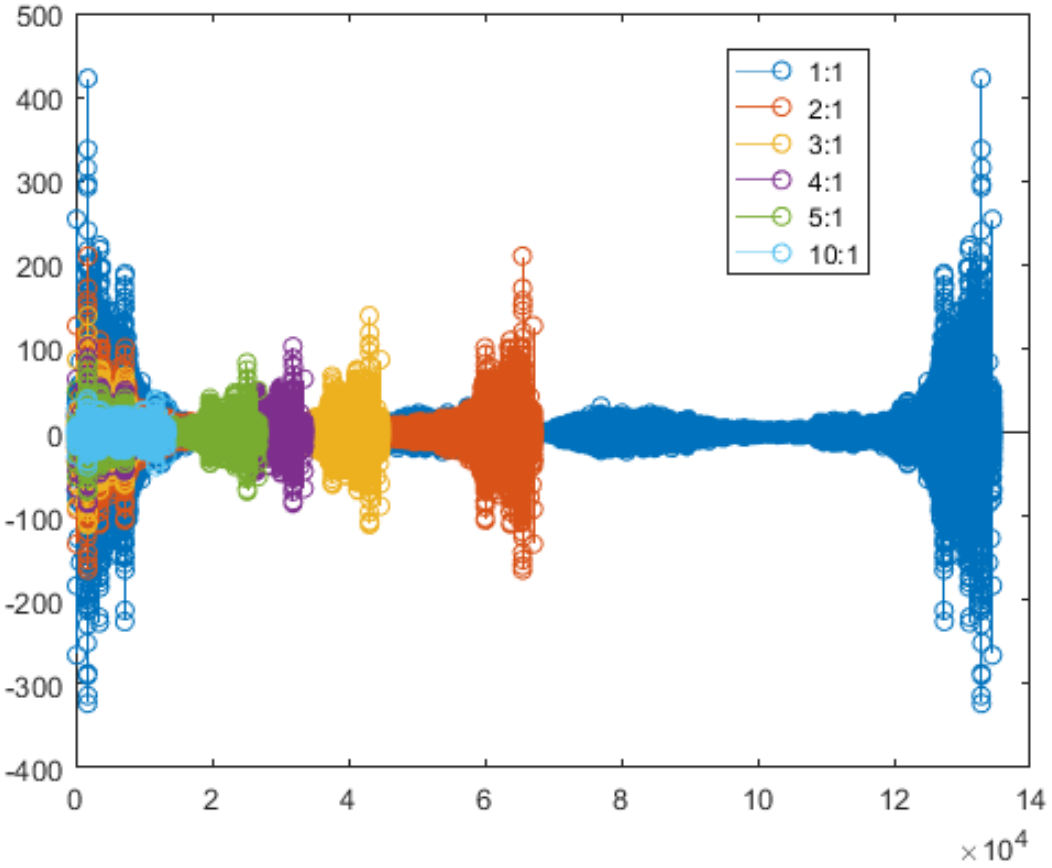


Figure 6: FFT plot of the sub-sampled audio clip

Table 2 below lists all the observations made when the sub-sampled audio was played.

Table 2: Observations made by playing sub-sampled audio clip

Sub-Sample ratio	Ratio
2:1	The sound is like the original with some minor losses in the audio
3:1	This sound is also like the original, but with more noise than the observations made with 2:1 ratio
4:1	This sound has more noise compared to the previous 2 ratios than, with the sound almost resembling a muffled voice
5:1	This sound is more muffled than before with the audio almost not hearable in low volume
10:1	This sound has the most losses, having the greatest muffled audio. Only a little bit of the audio could be understood

g) The following results were obtained after quantization. The MATLAB code used to achieve quantization can be seen in Figures 28 and 29 in the appendix:

1-bit quantization:

The graph in Figure 7 below shows the discrete plot of the 1-bit quantized signal. In this case, significant noise could be heard along with the audio clip, the noise occurring due to quantization

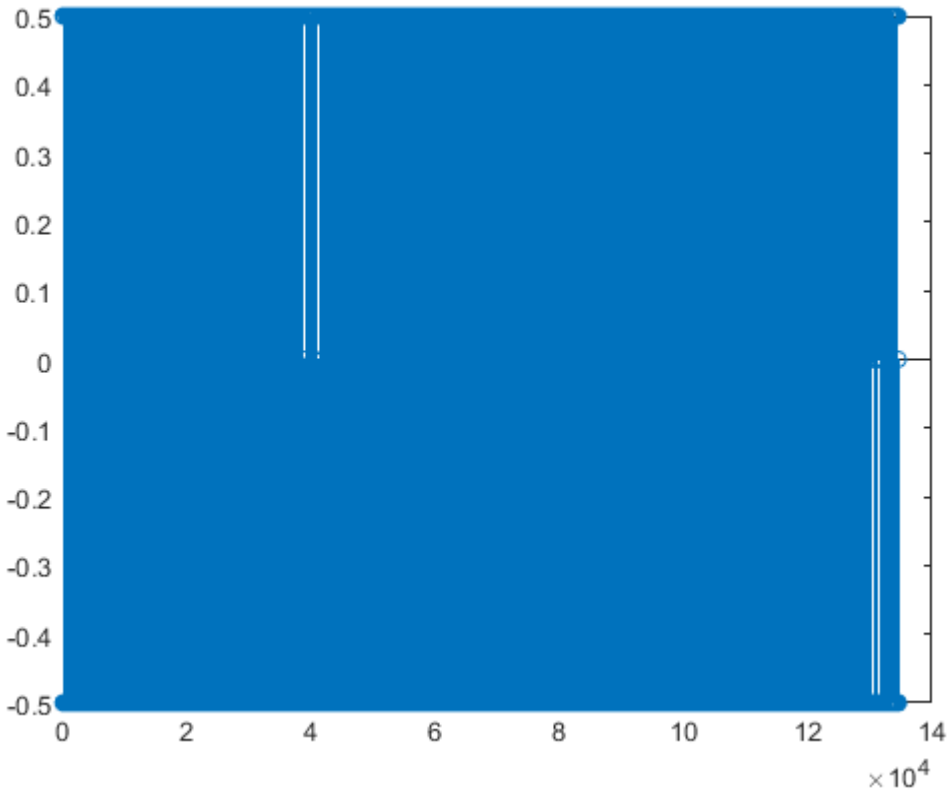


Figure 7: 1- bit quantization

2-bit quantization:

The graph in Figure 8 below shows the plot for the 2-bit quantized signal. In this case, the signal could be heard along with background noise, severely hampering the quality. However, the sound quality was noticeably better compared with 1-bit quantization.

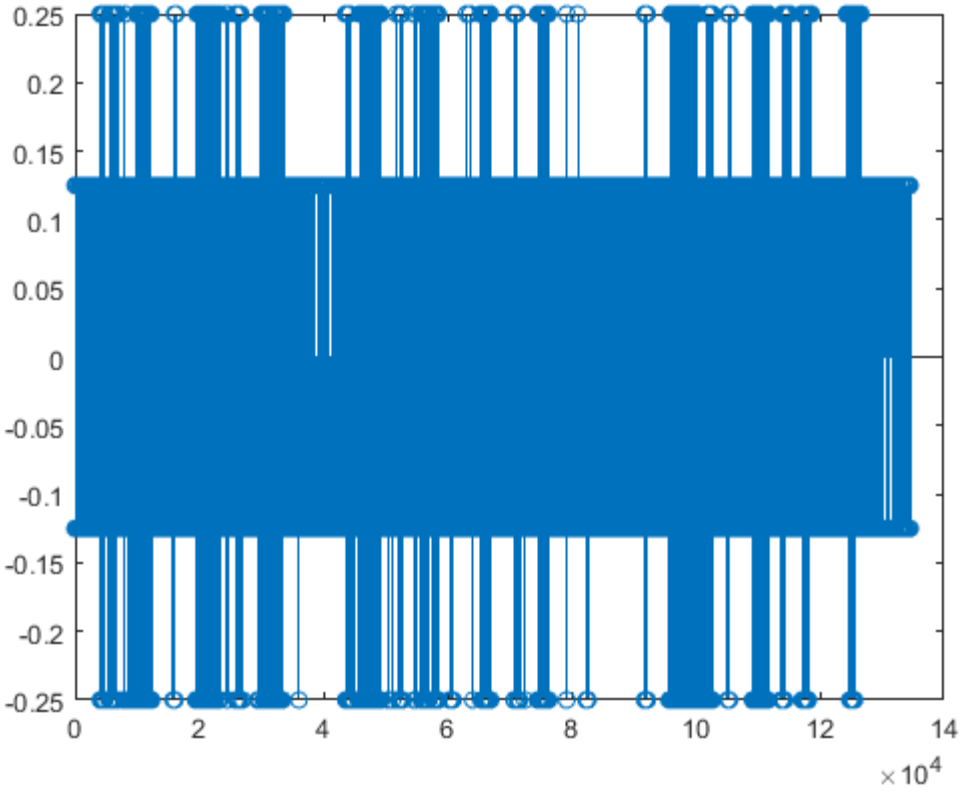


Figure 8: 2-bit quantization

4-bit quantization:

The graph for 4-bit quantization is as shown in Figure 9 below. In this case, the original signal could be heard along with noise, however the quality was much better compared with 1-bit and 2-bit quantization with the noise intensity seemed lower than 1-bit and 2-bit quantization noise.

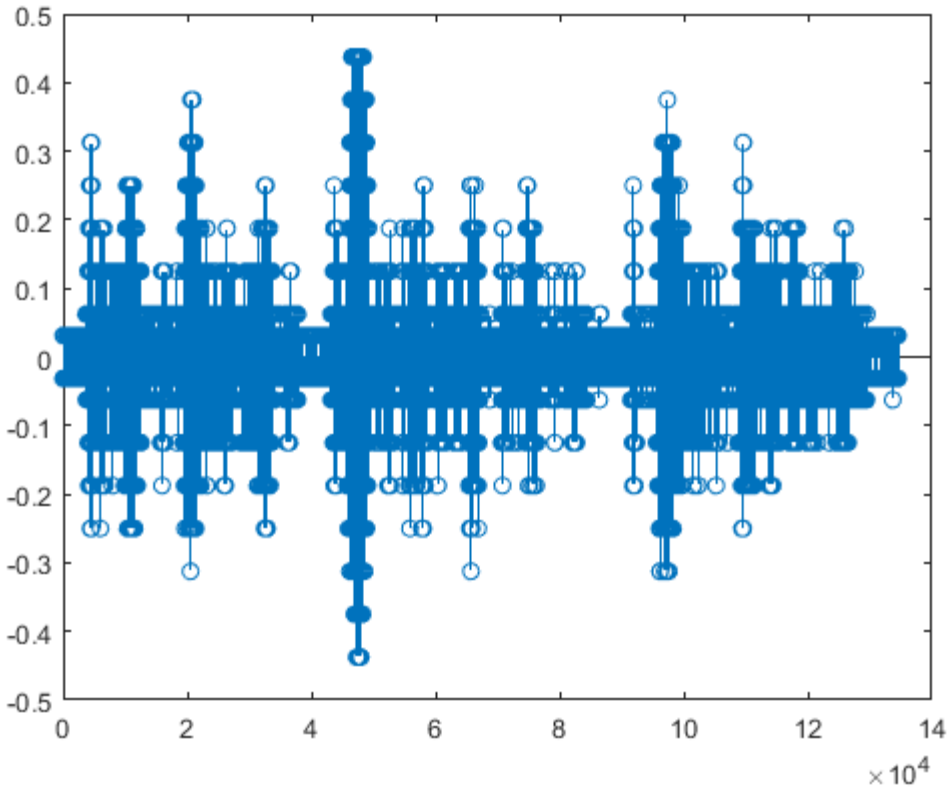


Figure 9: 4-bit quantization

8-bit quantization:

In this case the sound was very clear, almost close to the original. Some low and minor noise could be heard with the signal at very specific points and not throughout the audio playback, as was the case with 4-bit quantization. The discrete plot is shown in Figure 10 below, which is almost the same as the discrete plot of the original signal.

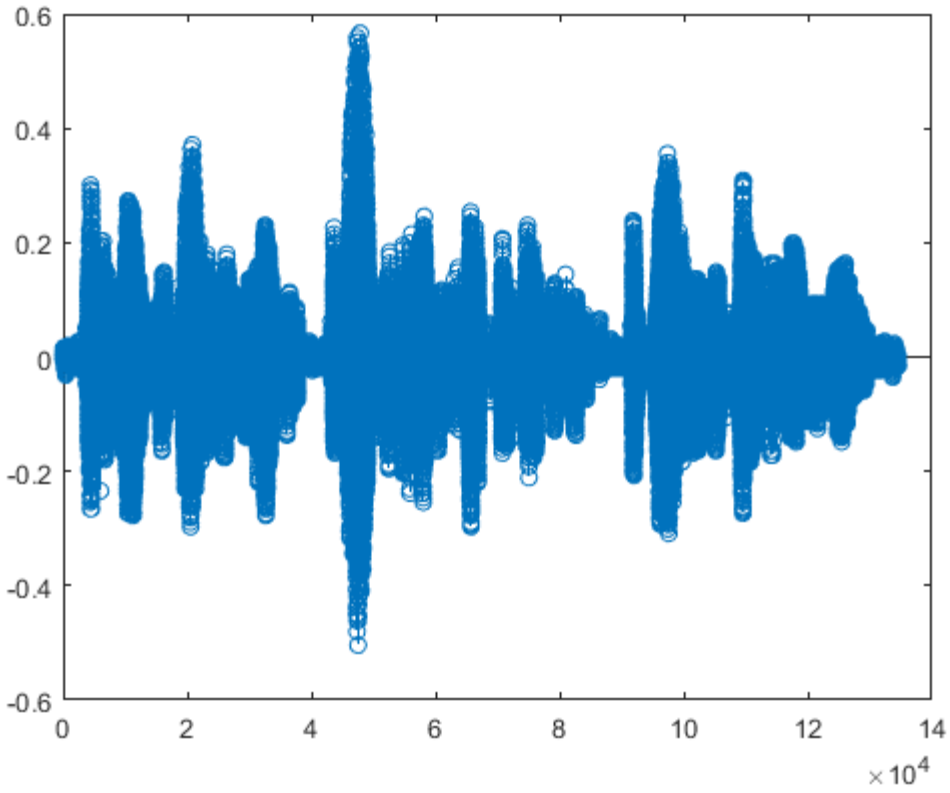


Figure 10: 8-bit quantization

16-bit quantization:

In this case the sound quality was the same as the original and no noise could be heard. The plot is shown in Figure 11 below, which is now closer to the original and better in shape compared to the plot of 8-bit quantization.

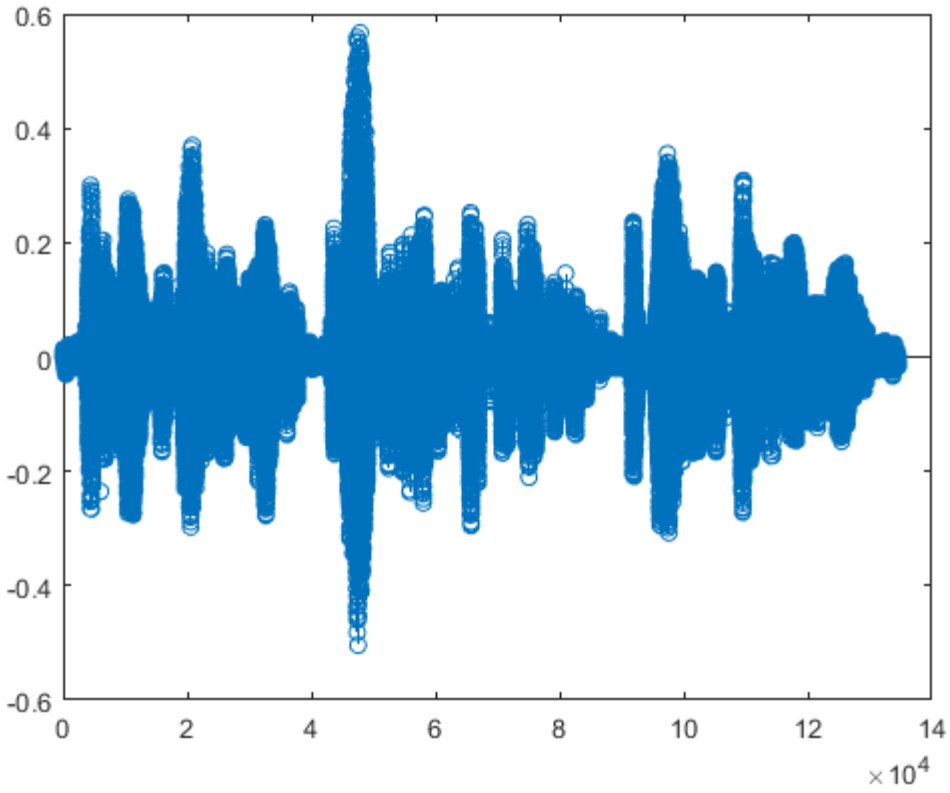


Figure 11:16-bit quantization

Question 4

a) Refer to Figure 31 in Appendix for the filter_demo block diagram used in this question and Figure 32 in Appendix for the Verilog script used to quantize the audio. A symbol was made for script and used in filter_demo block diagram.

b) Refer to Figure 33 in Appendix for the Verilog script used achieve aliasing. A symbol was made for script and used in filter_demo block diagram shown in Figure 31, as before.

d) Figure 34 in appendix shows the block diagram of the uniform additive Gaussian noise samples. 6 units of the provided uniform generator module was used here.

e) In MATLAB, the rand function can be used to generate uniformly distributed numbers such as:

$$a + rand(n, 1) * (b - a)$$

where a is the minimum, b is the maximum and n is the number of random number generated in vector.

In our case, where the interval is [-1 1] and we are adding 6 uniform generators, MATLAB script to show the noise generated is in fact close to being Gaussian distributed is shown in Figure 12:

```
function unifrom_gaussian_adder

uniform_noise = [];
for k = 1:1000
        uniform_noise =  [sum(-1 + rand(6,1)*2)    uniform_noise];
end
figure
hist(uniform_noise)

end
```

Figure 12: MATLAB script for generating uniform Gaussina noise samples

The plot of this script in Fig 13 shows that the noise generated with six uniform number generators has a Gaussian distribution.

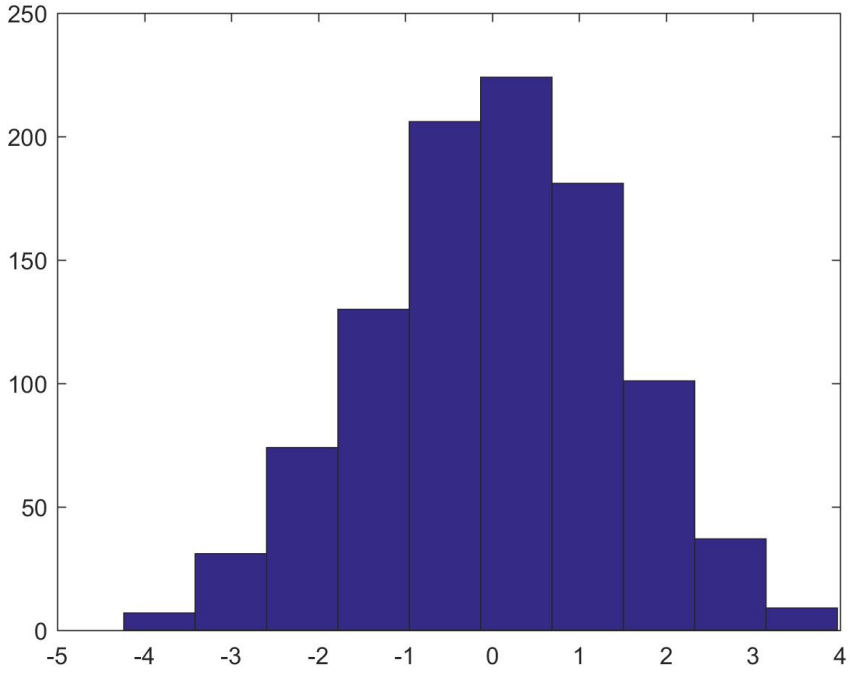


Figure 13: Plot of uniform Gaussian noise

f) The variance and mean of the noise produced by adding 6 consecutive uniformly generated samples is:

$$\text{variance} = 6\sigma$$

$$\text{mean} = \mu \text{ (in this case it's 0)}$$

The central limit theorem states that the sampling distribution of any random, independent variable (uniform samples) of same interval will be normally distributed for large sample sizes. This theorem applies as long as the random samples are on same interval (size).

g) The noise power was adjusted by masking most significant bits in noise. Estimating the each masked bit corresponds to 6dB and that SNR of 0dB is such that the most significant 4 bits of noise are shifted to the right. The different SNR values were obtained as shown in Table 3:

Table 3: SNR values with their corresponding bit shifts

SNR /dB	Right Shift in Added Gaussian Noise
-10	Shifted two bits
0	Shifted four bits
10	Shifted six bits
20	Shifted seven bits
30	Shifted nine bits
40	Shifted eleven bits

Refer to Figure 35 in appendix for the filter_demo block diagram used for corrupting the audio signal with a selected SNR level. The SNR levels were adjusted via volume switches. Refer to Figure 36 in appendix for the Verilog script used to shift the Gaussian noise bits.

Question 5

a) Here is an example of 4 equations with 4 unknowns where the real and mod2 solutions differ

$$1a + 0b + 1c + 1d = 1$$
$$1a + 1b + 1c + 1d = 0$$
$$0a + 1b + 1c + 0d = 1$$
$$1a + 1b + 1c + 0d = 0$$

Solving the above four equations yields the following results in base-10: a = -1, b = -1, c = 2 and d = 0. Solving the equations in mod-2 gives the following: a = 1, b = 1, c = 0 and d = 0.

b) The following example shows 4 linear equations with 4 unknowns where the real and mod-2 solutions differ but only contain 0 and 1:

$$1a + 1b + 0c + 0d = 0$$
$$0a + 1b + 0c + 1d = 1$$
$$1a + 0b + 1c + 0d = 0$$
$$1a + 0b + 0c + 1d = 1$$

Solving the above four equations yields the following results in base-10: a = 0, b = 0, c = 0 and d = 1. Solving the equations in mod-2 gives the following: a = 1, b = 1, c = 1 and d = 0.
This difference occurs when A matrix in base-10 is invertible but in mod-2 it's not
.

c) The above examples prove that it is necessary to implement our own function to solve any given system of linear equations in binary. The approach of solving the equation in real(base-10) domain and then converting the answer to binary does not always work, as the above example

has proven. This is a necessary step during the implementation of erasure decoder for binary error control.

d) Refer to Figures 37 and 38 for the implementation on the function "binSolve" which uses Gaussian elimination to solve 4 linear equations with 4 unknowns in mod-2.

Question 6

a) A system is said to be causal if its output depends only on present and past input values, and not future inputs. For discrete LTI systems, all outputs are defined via the convolutional theorem:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

For non-zero h[n] when n < 0, the summation would expand as such:

$$y[n] = x[-n]h[2n] + x[-n+1]h[2n-1] + \cdots x[-1]h[n+1] + x[0]h[n] + x[1]h[n-1] + \cdots x[n-1]h[1] + x[n]h[0] + x[n+1]h[-1] + x[n+2]h[-2] + \cdots$$

It can be noticed from the last two terms of the summation that when h[n] is negative the output depends on future values of x (x[n+1], x[n+2] and so on). This goes against the definition of causality and hence the LTI system is not causal if impulse response is non-zero for n < 0.

b) For BIBO stability, any bounded input, i.e. $|x[n]| < \infty$, should give a bounded output, i.e. $|y[n]| < \infty$. For LTI systems, a system is not BIBO stable if $|h[n]|$ is not stable. This can be proved via the convolutional theorem, where assume that summation of $|x[n]| = A$, a finite constant:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

$$y[n] = \sum_{k=-\infty}^{\infty} |x[k]||h[n-k]|$$

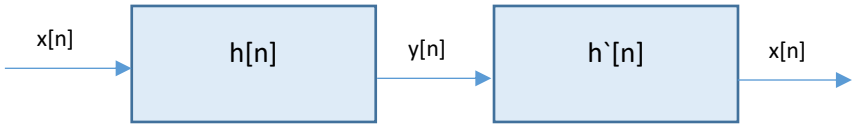
$$y[n] = \sum_{k=-\infty}^{\infty} A|h[n-k]|$$

$$y[n] = A\sum_{k=-\infty}^{\infty} |h[n-k]|$$

$$y[n] = A(\infty) = \infty$$

Therefore, the system is not stable.

c) For a system to be invertible, there should exist an impulse response such that when the output signal of the system is multiplied (or convoluted) with the inverse, the original input is recovered:



The following equation shows the output for the echo system used in Question 1. The echo is generated by delayed response of the input signal, modelled by the term x[n – A], where A is a real, positive constant.

$$y[n] = x[n] + x[n - A]$$

Converting this equation discrete time Fourier transform:

$$Y(e^{jw}) = X(e^{jw}) + e^{-jwA}X(e^{jw})$$

$$Y(e^{jw}) = X(e^{jw})(1 + e^{-jwA})$$

$$\frac{Y(e^{jw})}{X(e^{jw})} = H(e^{jw}) = 1 + e^{-jwA}$$

The invertible system is defined by:

$$H`(e^{jw}) = 1/H(e^{jw}) = 1/(1 + e^{-jwA})$$

Hence the system is only invertible when the term $1 + e^{-jwA}$ is non-zero, thus making $H`(e^{jw})$ real. Due to this condition, the system is not strictly invertible.

d) In this case, the impulse response of the system will be 1 for certain frequencies only. Assume that the impulse response is given as follows, where A is a real, positive constant:

$$h[n] = \delta[n] + \delta[n - A]$$

The z-transform for this impulse response is the following:

$$H(z) = 1 + z^{-A}$$

The inverse z-transform of the impulse above is:

$$H`(z) = 1/(1 + z^{-A})$$

Since the z-transform of the impulse defines the echo system, the inverse z-transform defines the echo removal system. Multiplying the sound with echo (in z-domain) with H`(z) should give the original sound back. The block diagram for this system, which would achieve echo removal, is shown in Figure 14 below.

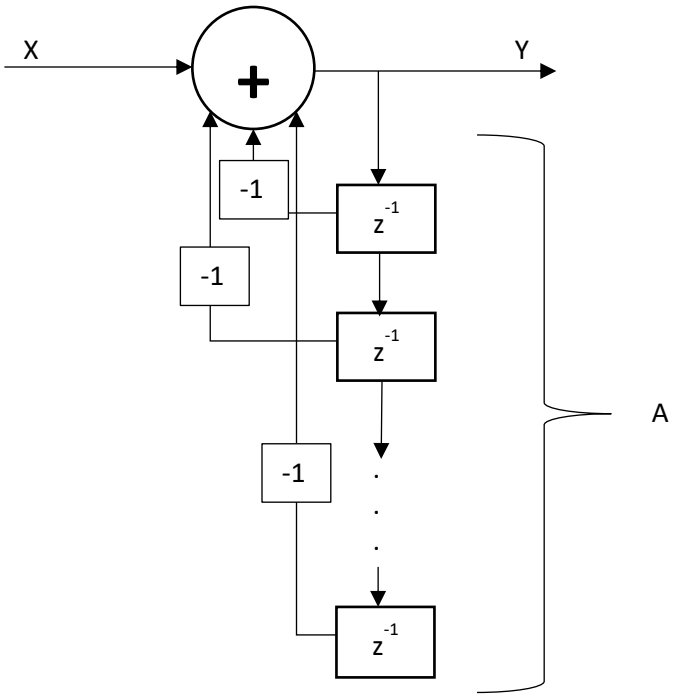


Figure 14: Echo removal block diagram

e) The convolution theorem states the following for LTI systems:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

It is also known that an LTI system has h[n] as its output when the input is $\delta[n]$. This can be stated as:

$$\delta[n] \rightarrow h[n]$$

Due to time-invariance we can say:

$$\delta[n-k] \rightarrow h[n-k]$$

Due to linearity:

$$x[k]\delta[n-k] \rightarrow x[k]h[n-k]$$

Again, due to linearity:

$$\sum_{k=-\infty}^{\infty} x[k]\delta[n-k] \rightarrow \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

Hence, the above states and proves the convolution theorem:

$$x[n] \rightarrow y[n]$$

f) The Fourier transform for any signal x[n] is defined by the following equality:

$$X(e^{jw}) = \sum_{n=-\infty}^{\infty} x[n]e^{-jwn}$$

The convolutional property states the following:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

Therefore, $Y(e^{jw})$ is given as such:

$$Y(e^{jw}) = \sum_{n=-\infty}^{\infty} y[n]\, e^{-jwn} = \sum_{n=-\infty}^{\infty} \left(\sum_{k=-\infty}^{\infty} x[k]h[n-k] \right) e^{-jwn}$$

$$Y(e^{jw}) = \sum_{n=-\infty}^{\infty} \left(\sum_{k=-\infty}^{\infty} x[k]h[n-k] \right) e^{-jw(n-k)}(e^{-jwk})$$

$$Y(e^{jw}) = \sum_{k=-\infty}^{\infty} x[k]e^{-jwk} \sum_{n=-\infty}^{\infty} h[n-k]\, e^{-jw(n-k)}$$

It can be observed that the two summations above correspond to the definition of discrete Fourier transform. Thus:

$$Y(e^{jw}) = X(e^{jw})H(e^{jw})$$

g) Aliasing refers to the sub-sampling of a signal. With the aid of aliasing, a continuous signal can be sampled at different frequencies to arrive at its close approximation in discrete time. A sample taken at every other interval will be a closer approximation to a sample taken at every 3rd interval. The plots below show the effect of aliasing by sampling a sine waveform (as shown in Figure 15) from continuous time domain to discrete time domain. In Figure 16 the sample is taken at every 0.1 seconds. In Figure 17 the sample is taken at every 0.3 seconds. In both figures, some points from the original waveform are missing due to aliasing.

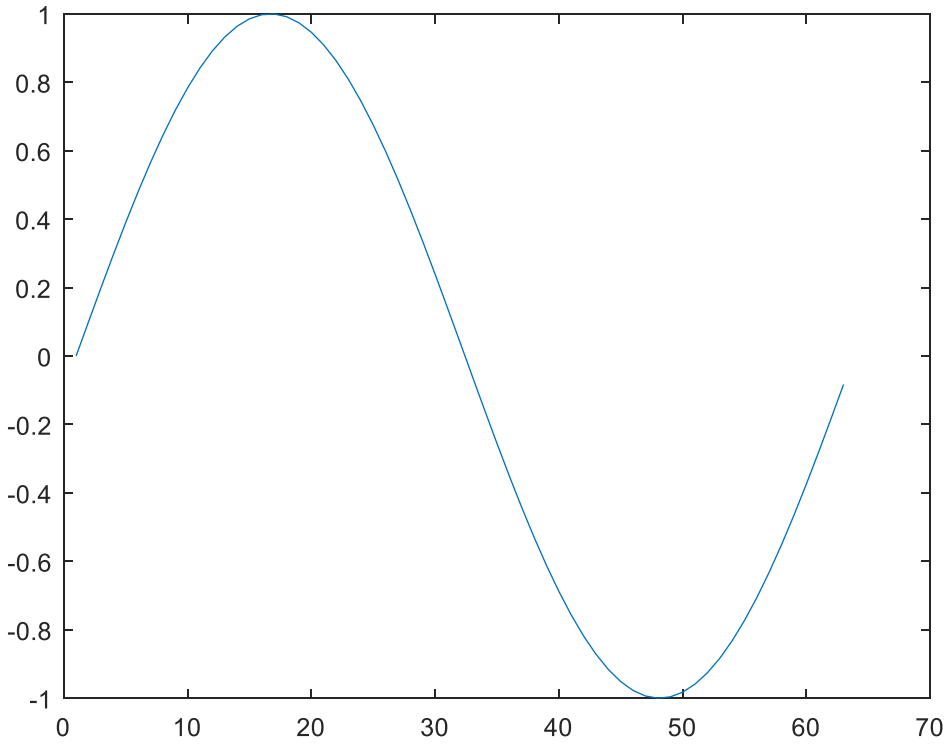


Figure 15: Sine plot in continuous time domain

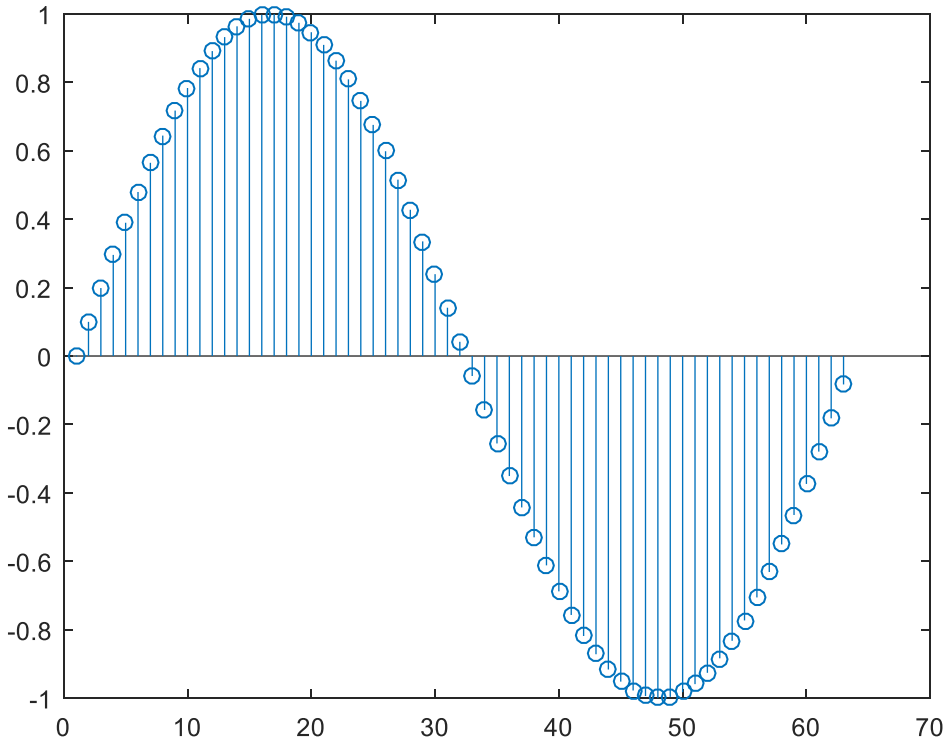


Figure 16: Sine function in discrete time domain, sample taken at every 0.1 seconds

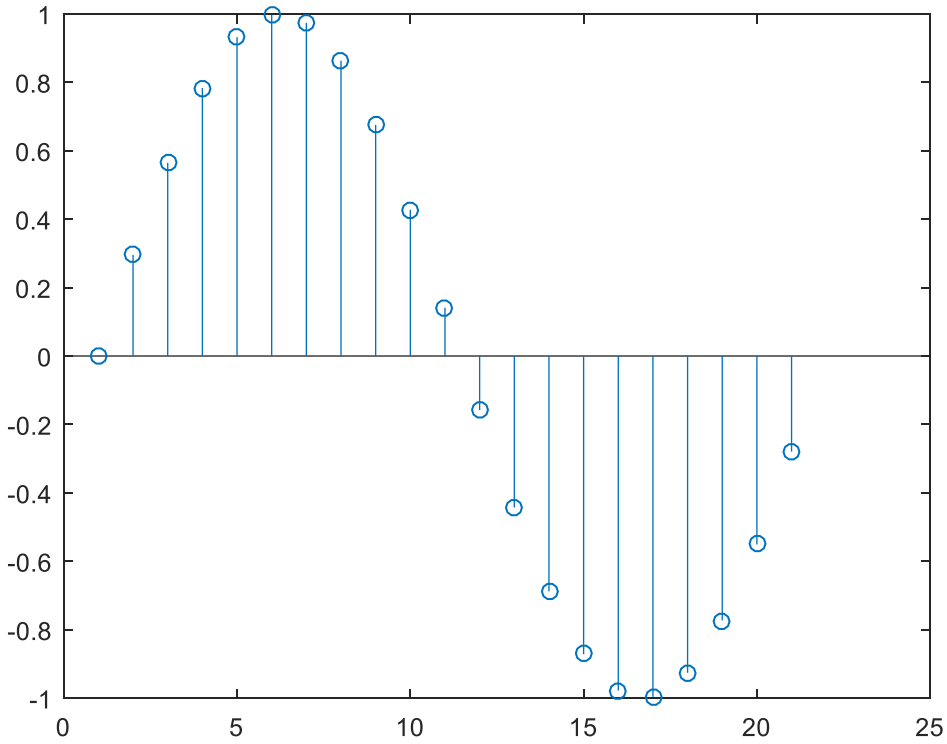


Figure 17: Sine function in discrete time domain, sample taken at every 0.3 seconds

Appendix

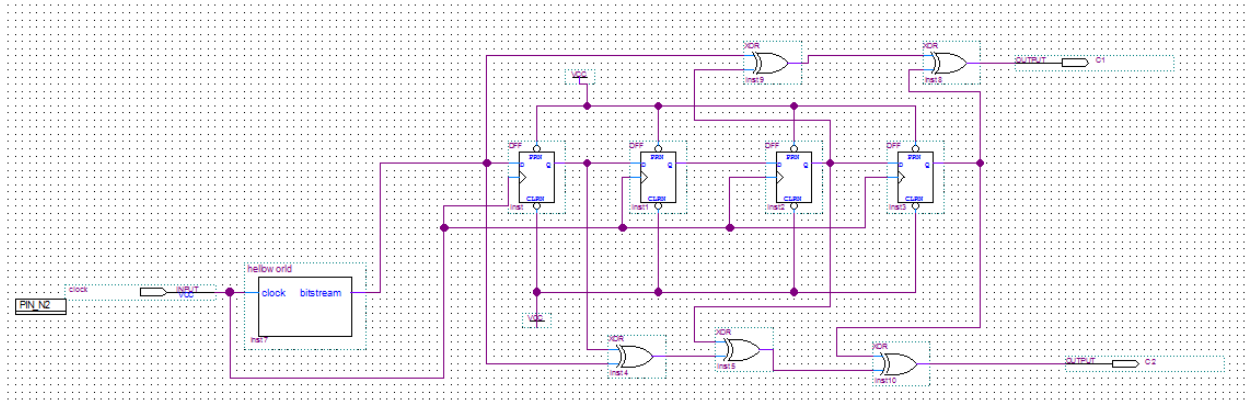


Figure 18: rate-half encoder

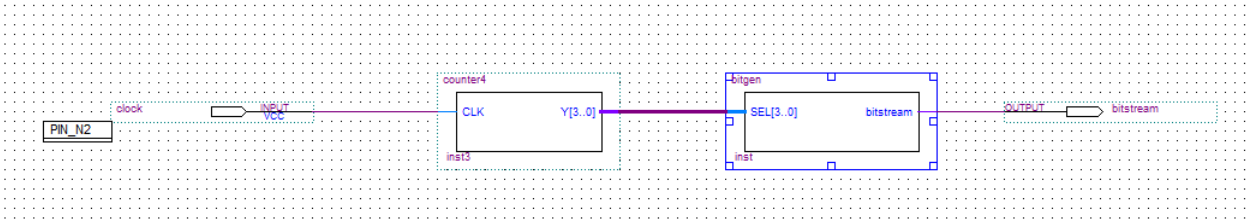


Figure 19: bit stream generator for rate half encoder

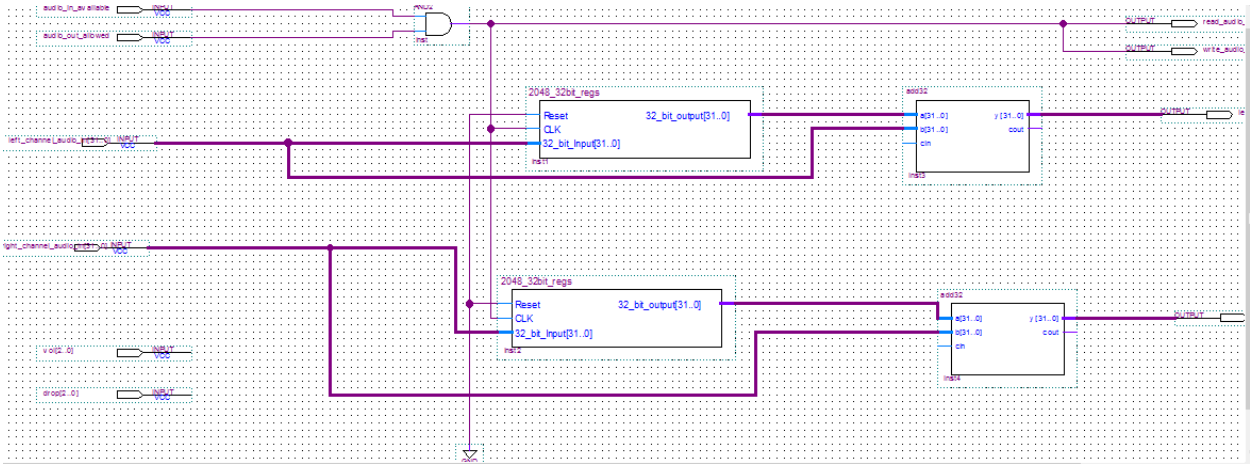


Figure 20: filter_demo for echo system

```
function decodedOutput = ViterbiDecoder(receivedRow)
%clear all
clc
%% Knowns for the (8,4,4) code
codebookEdges = [ 0 0 ; 1 1; 0 1; 1 0];
levelEdgeDistance = [];
numberOfPaths = [ 1 1 1 1; 2 2 2 2; 2 2 2 2; 1 1 1 1] ;

%% Calculate the distance at edges
% The distances are stored in a 4 by 4 vector where the rows are stages of
% trellis and columns are the edges
    for k = 1 : 4 % number of stages in trellis for (8,4,4) Hamming Code
        for n = 1 : 4 % number of edge calcs in each level
            if( (k == 2) | (k == 3))
                levelEdgeDistance(k,n) = (receivedRow(k+1)  - codebookEdges(n, 1))^2 +(receivedRow(k+3)  - codebookEdges(n, 2))^2;
            elseif (k ==4)
                levelEdgeDistance(k,n) = (receivedRow(k+3)  - codebookEdges(n, 1))^2 +(receivedRow(k+4)  - codebookEdges(n, 2))^2;
            else
                levelEdgeDistance(k,n) = (receivedRow(k)  - codebookEdges(n, 1))^2 +(receivedRow(k+1)  - codebookEdges(n, 2))^2;
            end
        end
    end
```

Figure 21: Viterbi decoder 1

```
%% Store edges distance in vectors for each stage
% Each stage contains a varying number of edges, thus a vector for each
% stage stores distance at each edge
    StageOneMinDist   = [];
    StageTwoMinDist   = [];
    StageThreeMinDist = [];
    StageFourMinDist  = [];

    for l = 1 : 4
        StageOneMinDist = [StageOneMinDist ,levelEdgeDistance(1,l)];
        if((l == 1) | (l == 2))
            StageTwoMinDist = [StageTwoMinDist, levelEdgeDistance(2, 1)];
            StageThreeMinDist = [StageThreeMinDist, levelEdgeDistance(3,1)];
            StageTwoMinDist = [StageTwoMinDist, levelEdgeDistance(2, 2)];
            StageThreeMinDist = [StageThreeMinDist, levelEdgeDistance(3,2)];
        else
            StageTwoMinDist = [StageTwoMinDist, levelEdgeDistance(2, 3)];
            StageThreeMinDist = [StageThreeMinDist, levelEdgeDistance(3,3)];
            StageTwoMinDist = [StageTwoMinDist, levelEdgeDistance(2, 4)];
            StageThreeMinDist = [StageThreeMinDist, levelEdgeDistance(3,4)];

        end
            StageFourMinDist = [StageFourMinDist, levelEdgeDistance(4, 1)];
    end
```

```
%% Search Step
% ADD-STORE-COMPARE at each state/stage of trellis taking the minimum sum
% and storing it in  a vector at each stage for each state
for stage = 1 : 4
    for state = 1 : 4
        if (numberOfPaths(stage,state) == 1)
            if(stage == 1)
                shortestPathsVect(stage,state) =  StageOneMinDist(state);
                shortestPathCodeword(stage, state) = numberOfPaths(stage,state);
            else
                if (shortestPathCodeword(2 , state) == shortestPathCodeword(3 , state))
                    shortestPathsVect(stage,state) =  StageFourMinDist(state) + shortestPathsVect(stage - 1,state);
                    shortestPathCodeword(stage, state) = numberOfPaths(stage,state);
                else
                    if(state == 1 | state == 3)
                        shortestPathsVect(stage,state) =  StageFourMinDist(state + 1) + shortestPathsVect(stage - 1,state);
                    else
                        shortestPathsVect(stage,state) =  StageFourMinDist(state - 1) + shortestPathsVect(stage - 1,state);
                    end
                    shortestPathCodeword(stage, state) = numberOfPaths(stage,state);
                end
            end
        end
        if (numberOfPaths(stage,state) == 2)
            if(stage == 2)
                    minDistSum = [StageTwoMinDist(2*state - 1) + shortestPathsVect(stage - 1,state); StageTwoMinDist(2*state) + shortestPathsVect(stage
                    for s =1 : size(minIndex)
                        shortestPathsVect(stage,state) =  minDistSum(minIndex(s));
                        shortestPathCodeword(stage,state) = minIndex(s);
                    end
            else
                    minDistSum = [StageThreeMinDist(2*state - 1) + shortestPathsVect(stage - 1,state); StageThreeMinDist(2*state) + shortestPathsVect(st
                    minIndex = find(minDistSum == min(minDistSum(:)));
                    for s =1 : size(minIndex)
                        shortestPathsVect(stage,state) =  minDistSum(minIndex(s));
                        shortestPathCodeword(stage,state) = minIndex(s);
                    end
            end
        end
    end
end
```

Figure 22: Viterbi decoder 2

```
%% Find min sum of the four states
% Take the path of that state

shortestPaths = shortestPathsVect(4, :);
CodewordPath = [];

minPath = find(shortestPaths == min(shortestPaths(:)));

for h = 1: size(minPath)
    CodewordPath = [CodewordPath; shortestPathCodeword(:,minPath(h))];
end
```

Figure 23: Shortest path finder

```
% part b
x_conv = conv(x, h);

sound(x, 16000);
sound(h, 16000);
sound(x_conv, 16000);
```

Figure 24: Convolution of impulse with audio

```
% part c
% shifted h => append col of zeros by shifted amout before h
h_shift = [zeros(3000,1);h];
h_echo = [h;zeros(3000,1)] + h_shift;
x_conv = conv(x, h_echo);
sound(x_conv, 16000);
```

Figure 25: echo sound generator

```
% part d
x_flip = flipud(x);
sound(x_flip, 16000);
```

Figure 26: Playing the sound in reverse

```
% part f
x_sub_2 = [];

for i = 1 : length(x)
    if (mod(i, 2) == 0)
        x_sub_2 = [x_sub_2; x(i)];
    end
end

x_sub_3 = [];
for i = 1 : length(x)
    if (mod(i, 3) == 0)
        x_sub_3 = [x_sub_3; x(i)];
    end
end

x_sub_4 = [];
for i = 1 : length(x)
    if (mod(i, 4) == 0)
        x_sub_4 = [x_sub_4; x(i)];
    end
end
x_sub_5 = [];
for i = 1 : length(x)
    if (mod(i, 5) == 0)
        x_sub_5 = [x_sub_5; x(i)];
    end
end

x_sub_10 = [];
for i = 1 : length(x)
    if (mod(i, 10) == 0)
        x_sub_10 = [x_sub_10; x(i)];
    end
end
```

Figure 27: Aliasing implementation

```
oneBitQuant = x;
for i = 1 : length(oneBitQuant)
    if (oneBitQuant(i) > 0)
        oneBitQuant(i) = 1;
    end
    if (oneBitQuant(i) < 0)
        oneBitQuant(i) = -1;
    end
end
oneBitQuant = oneBitQuant/2^1;
stem(oneBitQuant);
sound(oneBitQuant, 16000);

twoBitQuant = round(x*2^2);
for i = 1 : length(twoBitQuant)
    if (twoBitQuant(i) > (2^1 - 1))
        twoBitQuant(i) = 1;
    end
    if (twoBitQuant(i) < -(2^1 - 1))
        twoBitQuant(i) = -1;
    end

    if (x(i) < 0 && twoBitQuant(i) == 0)
        twoBitQuant(i) = -0.5;
    end

    if (x(i) >= 0 && twoBitQuant(i) == 0)
        twoBitQuant(i) = 0.5;
    end
end
twoBitQuant = twoBitQuant/2^2;
stem(twoBitQuant);
sound(twoBitQuant, 16000);
```

Figure 28: 1-bit and 2-bit quantization

```
fourBitQuant = round(x*2^4);
for i = 1 : length(fourBitQuant)
    if (fourBitQuant(i) > (2^3 - 1))
        fourBitQuant(i) = 7;
    end
    if (fourBitQuant(i) < -(2^3 - 1))
        fourBitQuant(i) = -7;
    end

    if (x(i) < 0 && fourBitQuant(i) == 0)
        fourBitQuant(i) = -0.5;
    end

    if (x(i) >= 0 && fourBitQuant(i) == 0)
        fourBitQuant(i) = 0.5;
    end
end
unique(fourBitQuant)
fourBitQuant = fourBitQuant/2^4;
stem(fourBitQuant);
sound(fourBitQuant, 16000);

EightBitQuant = round(x*2^8);
for i = 1 : length(EightBitQuant)
    if (EightBitQuant(i) > (2^8 - 1))
        EightBitQuant(i) = 127;
    end
    if (EightBitQuant(i) < -(2^8 - 1))
        EightBitQuant(i) = -127;
    end

    if (x(i) < 0 && EightBitQuant(i) == 0)
        EightBitQuant(i) = -0.5;
    end

    if (x(i) >= 0 && EightBitQuant(i) == 0)
        EightBitQuant(i) = 0.5;
    end

end
EightBitQuant = EightBitQuant/2^8;
stem(EightBitQuant);
sound(EightBitQuant, 16000);
```

Figure 29: 4-bit and 8-bit quantization

```
sixteenBitQuant = round(x*2^16);
for i = 1 : length(sixteenBitQuant)
    if (sixteenBitQuant(i) > (2^16 - 1))
        sixteenBitQuant(i) = 32767;
    end
    if (sixteenBitQuant(i) < -(2^16 - 1))
        sixteenBitQuant(i) = -32767;
    end

    if (x(i) < 0 && sixteenBitQuant(i) == 0)
        sixteenBitQuant(i) = -0.5;
    end

    if (x(i) >= 0 && sixteenBitQuant(i) == 0)
        sixteenBitQuant(i) = 0.5;
    end
end
sixteenBitQuant = sixteenBitQuant/2^16;
stem(sixteenBitQuant);
sound(sixteenBitQuant, 16000);
```

Figure 30: 16-bit quantization

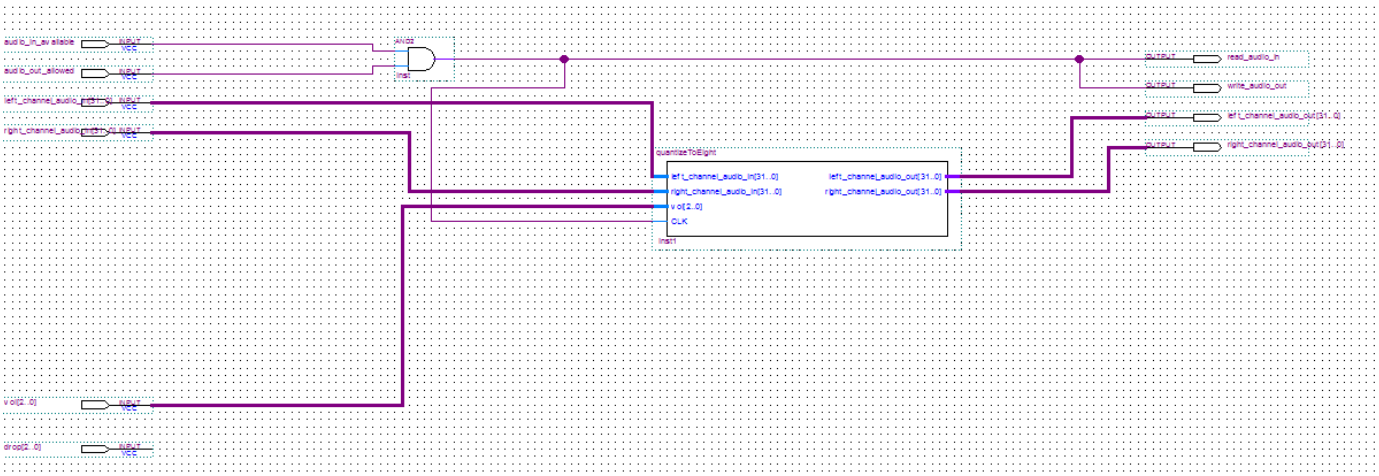


Figure 31: Filter_demo block diagram for Q4(a)

```
module quantizeToEight (left_channel_audio_in, right_channel_audio_in, left_channel_audio_out, right_channel_audio_out, vol, CLK);

   input [31:0] left_channel_audio_in, right_channel_audio_in;
   input [2:0] vol;
   output [31:0] left_channel_audio_out, right_channel_audio_out;
   input CLK;

   always@(posedge CLK)
   begin
   if(vol == 3'b001)
   begin
   left_channel_audio_out <= (left_channel_audio_in & 32'h80000000);
   right_channel_audio_out <= (right_channel_audio_in && 32'h80000000);
   end
   else if(vol == 3'b010)
   begin
   left_channel_audio_out = (left_channel_audio_in & 32'hC0000000);
   right_channel_audio_out = (right_channel_audio_in & 32'hC0000000);
   end
   else if(vol == 3'b011)
   begin
   left_channel_audio_out = (left_channel_audio_in & 32'hE0000000);
   right_channel_audio_out = (right_channel_audio_in & 32'hE0000000);
   end
   else if(vol == 3'b100)
   begin
   left_channel_audio_out = (left_channel_audio_in & 32'hF0000000);
   right_channel_audio_out = (right_channel_audio_in & 32'hF0000000);
   end
   else if(vol == 3'b111)
   begin
   left_channel_audio_out = (left_channel_audio_in & 32'hFF000000);
   right_channel_audio_out = (right_channel_audio_in & 32'hFF000000);
   end
   else
   begin
   left_channel_audio_out = left_channel_audio_in;
   right_channel_audio_out = right_channel_audio_in;
   end
   end

endmodule
```

Figure 32: Verilog script for audio quantization used in Q4(a)

```
module subSample(left_channel_audio_in, right_channel_audio_in, left_channel_audio_out, right_channel_audio_out, vol, CLK);

   input [31:0] left_channel_audio_in, right_channel_audio_in;
   input [2:0] vol;
   output [31:0] left_channel_audio_out, right_channel_audio_out;
   input CLK;
   integer clockCount = 0;

   always@(posedge CLK)
   begin

   clockCount = clockCount + 1;
   case(vol)

   3'b000 : begin
      // no aliasing
      left_channel_audio_out = left_channel_audio_in;
      right_channel_audio_out = right_channel_audio_in;
   end
   3'b001 : begin
      // 2 bit aliasing
      if (clockCount% 2 == 0)
      begin
      left_channel_audio_out = left_channel_audio_in;
      right_channel_audio_out = right_channel_audio_in;
      end
   end
   3'b010 : begin
      // 4 bit aliasing
      if (clockCount% 4 == 0)
      begin
      left_channel_audio_out = left_channel_audio_in;
      right_channel_audio_out = right_channel_audio_in;
      end
   end
   3'b100 : begin
      // 8 bit aliasing
      if (clockCount% 8 == 0)
      begin
      left_channel_audio_out = left_channel_audio_in;
      right_channel_audio_out = right_channel_audio_in;
      end
   end
   endcase
   end

endmodule
```

Figure 33: Verilog script used for aliasing for Q4(b)

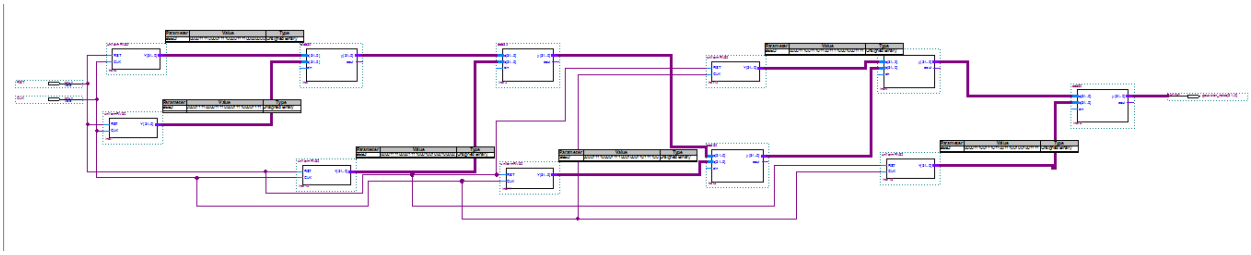


Figure 34: Block diagram for uniform Gaussian noise generator used for Q4(d)

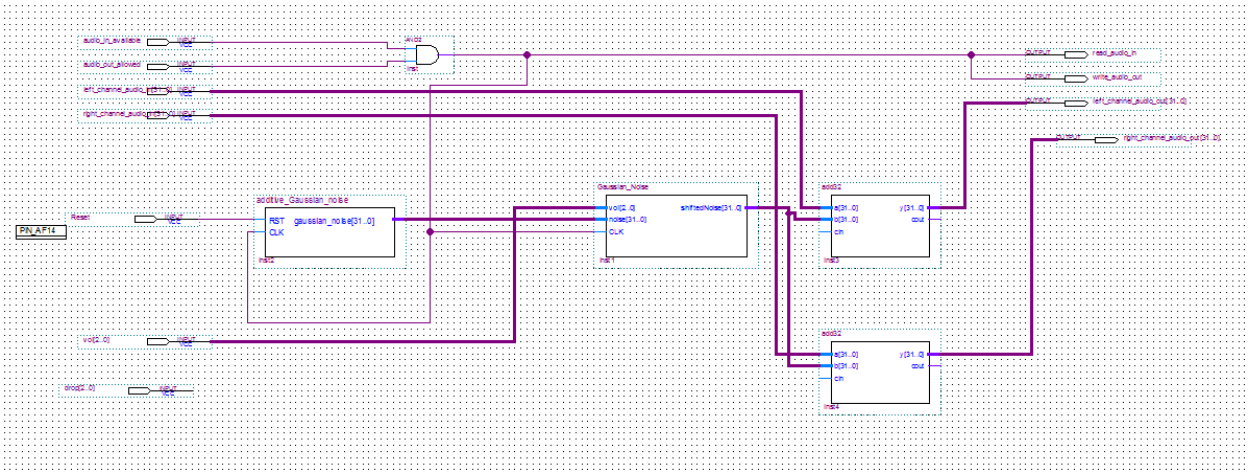


Figure 35: Filter_demo block diagram for corrupting audio signal with noise, used for Q4(g)

```
module Gaussian_Noise(vol, noise, shiftedNoise, CLK);
    input [2:0] vol;
    input [31:0] noise;
    output [31:0] shiftedNoise;
    input CLK;

    always@(posedge CLK)
    begin

case(vol)
    3'b001:  shiftedNoise = noise >>> 2;
    3'b010:  shiftedNoise = noise >>> 4;
    3'b100:  shiftedNoise = noise >>> 6;
    3'b101:  shiftedNoise = noise >>> 7;
    3'b110:  shiftedNoise = noise >>> 9;
    3'b111:  shiftedNoise = noise >>> 11;

    endcase
end
endmodule
```

Figure 36: Verilog script used for shifting Gaussian noise according to SNR levels, used for Q4(g)

```
function X = binSolve(A, B)
% use gaussian elimination to solve the equations
% returns the solutions in x, a defines the coefficients and b the
% RHS constants
% Solves only 4 equations with 4 unknowns

% initialize for recursion
X = B;
% r and c are row and column count of A respectively
[r, c] = size(A);

% exit if we have a zero row or if two rows are the same (which would give
% a zero row eventually)
for i = 1 : r
    if (sum(A(i, :)) == 0)
        X = [inf; inf; inf; inf];
        B = [inf; inf; inf; inf];
        return;
    end

    for row2 = 1 : r
        if (all(A(i, :) == A(row2, :)) && (i ~= row2))
            X = [inf; inf; inf; inf];
            B = [inf; inf; inf; inf];
            return;
        end
    end
end
% set solutionFound to true to return final output and exit recursion only
% when each row in A has only one 1
solutionFound = true;
for row = 1 : r
    if (sum(A(row, :)) ~= 1)
        solutionFound = false;
    end
end

if (solutionFound)
    % when solutionFound is true, A resembles an identity matrix (except
    % its not arranged). So we arrange X according to identity matrix which
    % will be the final output from the function.
    for row = 1 : r
        for col = 1 : c
            if (A(row, col) == 1)
                X(col, 1) = B(row, 1);
            end
        end
    end
    % set B as X to avoid recursion issues and return
    B = X;
    return;
end

% Now perform Gaussian elimination
% strategy: reduce number of ones in some rows
% only do one operation per recursion to avoid complexity
operationDone = false;
for row = 1 : r
    if (operationDone)
        break;
    end
```

Figure 37: Mod-2 Gaussian elimination solver

```
    for row2 = 1 : r
        numberOfOnesInRow = sum(A(row , :));
        % Check if numberOfOnesInRow can be reduced by row addition
        if (sum(mod(A(row , :) + A(row2 , :), 2)) < numberOfOnesInRow && row ~= row2)
            % Add row2 to row
            A(row , :) = mod(A(row , :) + A(row2 , :), 2);
            B(row , :) = mod(B(row , :) + B(row2 , :), 2);
            % break and move on to next row
            operationDone = true;
            break;
        end
    end
end

% if nothing was done before, this means that any additions doesn't reduce
% the number of 1s in any row. So now we try to make any row very similar
% or exact to another by addition
if (~operationDone)
    for row = 1 : r
        if (operationDone)
            break;
        end
        for row2 = 1 : r
            if (operationDone)
                break;
            end
            for row3 = 1 : r
                comparedRow = (mod(A(row , :) + A(row2 , :), 2) == A(row3, :));
                if ((sum(comparedRow) == 4 || sum(comparedRow) == 3) && row ~= row2)
                    % Add row2 to row
                    A(row , :) = mod(A(row , :) + A(row2 , :), 2);
                    B(row , :) = mod(B(row , :) + B(row2 , :), 2);
                    % break and move on to next row
                    operationDone = true;
                    break;
                end
            end
        end
    end
end

% Recurse the function until we reach the identity matrix for row_echelon_A
% temp output
X = binSolve(A, B);
end
```

Figure 38: Mod-2 Gaussian elimination solver 2