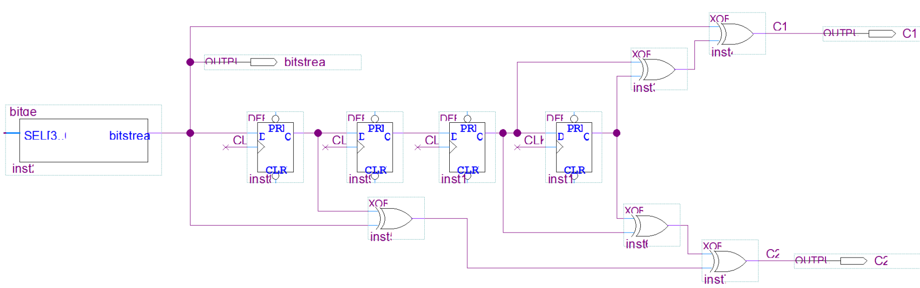
Signal Processing Lab 2

Malcolm Watt 260585950
David Lavoie-Boutin 260583602

November 10, 2016

1 Question 1

(a) We implemented the 1/2 convolution encoder using d-flip-flops as delay modules, and xor gates as modulo 2 adders. This results in the following circuit:

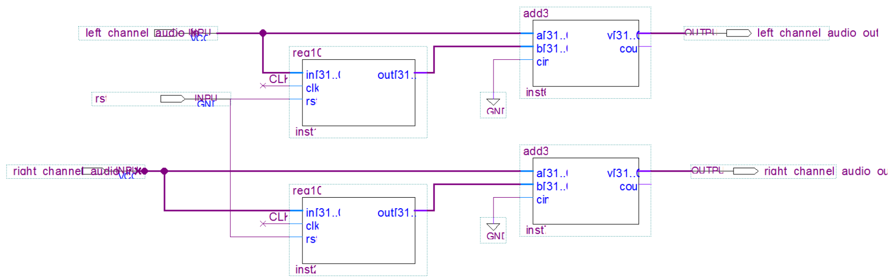


A full printout of the results, inspected with SignalTap is available in the appendix.

(b) To generate an echo using the FPGA board, we used a succession of d-flip-flops used as register to create a delayed copy of the samples, then used a 32 bits added to combine the real-time and the delayed signal, giving the illusion of an echo. The script 13 show our way of generating the circuit, and file 14 demonstrates the output for 10 gates.

2 Viterbi Decoding Algorithm for the Hamming (8,4,4) Code

A Hamming codeword c = (c1, c2, c3, c4, c5, c6, c7, c8) has been transmitted over an AWGN channel with noise mean being 0 and a positive variance. The received vector r = c + noise is observed to be r = (0.54, - 0.12, 1.32, 0.41, 0.63, 1.25, 0.37, -0.02).



Listing 1: 844 bruteforce decoder

```
%% Implement bruteforce solution for gaussian noise decoding
% Inline approximations of the # of floating point opperations
% Assume we consider each element of the matrix and ignore any
% optimizations that Matlab does behind the scenes
bin =  generate_binary_values(4);
vals = zeros(16,8);
diffs = zeros(16, 8);
r = [0.54 -0.12 1.32 0.41 0.63 1.25 0.37 -0.02];


for i = 1:length(bin)
    vals(i,:) = encoder_844(bin(i,:));
    diffs(i,:) = vals(i,:) - r; % 8 floating point subtractions
end % 8 * 16 = 128 floating point subtractions

% squaring: 128 multiplications
% assume cumulative: 7 per row --> 7*16 = 112 additions
% Min, assume cumulative: 7 comparisons
[M, I] = min(sum(diffs.^2, 2));

% total number of ops: 128 subs, 128 mults, 112 adds, 7 comps
result = vals(I,:)
```

2.1 Brute Force Approach

2.1.1 Decode r using codebook

Using Matlab, the code book of the (8,4,4) code and material discussed in the lecture/lab, determine the closest of all 16 codewords to the received r. Submit your script and the answer.

The closest code word $\hat{c}$ is (0 0 1 1 1 1 0 0) which means the decoded message $\hat{r}$ is (0 0 1 1). The script can be seen below. The generate_binary_values_function and the encoder_844 function can be found in the Matlab Appendix.

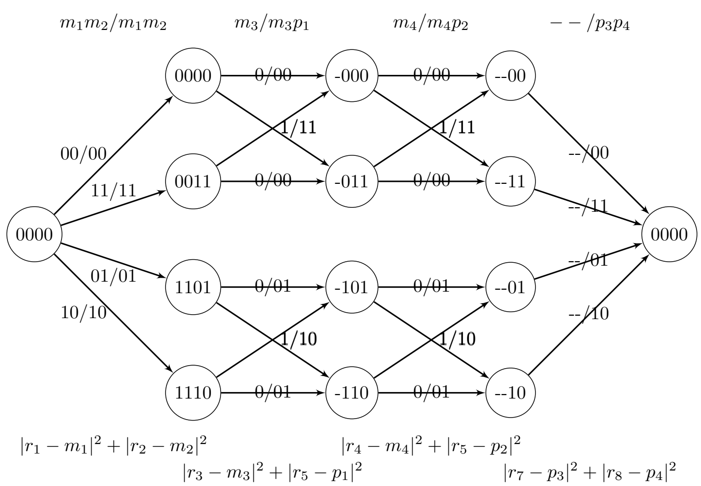


Figure 1: Optimized trellis diagram for the (8, 4, 4) Hamming Code.

2.1.2 Flop count

How many floating point operations (additions, multiplications and comparisons) does the approach from part (a) use to decode a codeword?

In order to decode the codeword, we require 128 subtractions, 128 multiplications (for the squaring opperation), 112 additions, and 7 comparisons. This is a total of **375** floating point opperations to decode a single codeword.

2.2 Optimized Trellis Approach

2.2.1 Trellis Design

Design an optimized trellis diagram for this code that has at most 4 states per stage.

Figure 1 shows our optimized trellis diagram for the (8, 4, 4) code, with only 4 states per stage maximum. Note the edge weights for each stage are provided below the edges, while the inputs and outputs are provided above the edges.

2.2.2 Trellis Implementation in MatLab

Redo part (a1) in Matlab by using Viterbi decoding algorithm for the given Hamming codeusing its 4-state optimized trellis. Determine and implement the appropriate edge weights for individual trellis edges, then use the survivor path approach to determine your answer. Submit your stage-by-stage distance matrices and results of each stage survivor search.

The Viterbi decoder has two major steps:

1. Populate all path lengths

2. Grow paths and find minimum path

We should state right away that we did not use the reduced path lengths in our MatLab implementation because they then required a significant amount of conditional logic to determine where to place each of the paths. Since MatLab optimizes basic calculations anyway it seemed unnecessary.

As far as the path growth is concerned, we applied the symmetric methodology proposed in the lab, finding the shortest path from the start to each of the 4 center nodes and then the shortest paths from the end to the 4 center nodes. Once this was done we combined paths to and from each node and compared the four of them to determine the shortest path.

Our resulting vector was the same as with our brute force decoder.

Our implementation for this part is the exact same as the one in section 2.3.1, so we do not show it twice.

2.2.3 Viterbi Flop Count

To generate each of the edges it took 3 flops, so all edges it took 48 flops. This is where a lot of the reduction would normally occur in a reduced implementation.

The first reduction required two flops per path, with there being 8 paths, so there were 16 flops. Again, this could be optimized by using the fact that the messages are symmetric, but it requires a fair amount of conditional logic so the tradeoff isn't really worth it in MatLab.

The second reduction required the sum of the left and right paths and 3 comparisons. This is the only part that is fully optimized in our MatLab implementation.

Our total flop count is 71 flops, which is much higher than the fully optimized system. Which can go as low as 21.

2.3 Viterbi Decoder for General r

2.3.1 Matlab Implementation

The only real difference between the script above and the implementation for a general r is that the general r is wrapped in a function.

Listing 2: 844 Viterbi decoder for general r

```
function [ shortest_path_bin ] = two_c_one(r)
%This function decodes the given (8,4,4) codeword using Viterbi
    decoding.

% This is the indicator for each of the tfs that is made
tfs = [
    0 0 0 0 ;
    3 3*2^2 3*2^4 3*2^6 ;
    1 1*2^2 1*2^4 1*2^6 ;
    2 2*2^2 2*2^4 2*2^6 ;
];

```

```
% First state transition: 4 edges
top_s1 = (r(1))^2 + (r(2))^2;
top_s1_2 = (r(1)-1)^2 + (r(2)-1)^2;
bottom_s1 = (r(1)-1)^2 + (r(2))^2;
bottom_s1_2 = (r(1))^2 + (r(2)-1)^2;

edges_stage1 = [ top_s1 top_s1_2 bottom_s1 bottom_s1_2 ]';

% Second state transition
stay_top_s2 = (r(3))^2 + (r(5))^2;
stay_top_s2_2 = (r(3)-1)^2 + (r(5)-1)^2;
stay_bottom_s2 = (r(3)-1)^2 + (r(5))^2;
stay_bottom_s2_2 = (r(3))^2 + (r(5)-1)^2;

edges_stage2 = [ stay_top_s2 stay_top_s2_2 stay_bottom_s2 stay_bottom_s2_2 ]';

% Third state transition
stay_top_s3 = (r(4))^2 + (r(6))^2;
stay_top_s3_2 = (r(4)-1)^2 + (r(6)-1)^2;
stay_bottom_s3 = (r(4)-1)^2 + (r(6))^2;
stay_bottom_s3_2 = (r(4))^2 + (r(6)-1)^2;

edges_stage3 = [ stay_top_s3 stay_top_s3_2 stay_bottom_s3 stay_bottom_s3_2 ]';

% Fourth state transition
top_s4 = (r(7))^2 + (r(8))^2;
top_s4_2 = (r(7)-1)^2 + (r(8)-1)^2;
bottom_s4 = (r(7)-1)^2 + (r(8))^2;
bottom_s4_2 = (r(7))^2 + (r(8)-1)^2;

edges_stage4 = [ top_s4 top_s4_2 bottom_s4 bottom_s4_2 ]';

% Full state of the map
grph = horzcat([edges_stage1 edges_stage2 edges_stage3 edges_stage4 ]);

% First reduction - Reduce the matrix by partitioning as follows
%
% a1 a2 | a3 a4          path to a | path from a
% b1 b2 | b3 b4          path to b | path from b
% ---------------   -->   --------   ----------
% c1 c2 | c3 c4          path to c | path from c
% d1 d2 | d3 d4          path to d | path from d
%
% and reducing each of the partitions into a path to it's middle nodes

% top left
top_left = grph(1:2,1:2);

top_left_tf = tfs(1:2,1:2);

paths_to_a = [
    top_left(1,1) + top_left(1,2);
    top_left(2,1) + top_left(2,2);
```

```
];

paths_to_b = [
    top_left(2,1) + top_left(1,2);
    top_left(1,1) + top_left(2,2);
];

bins_to_a = [
    top_left_tf(1,1) top_left_tf(1,2);
    top_left_tf(2,1) top_left_tf(2,2);
];

bins_to_b = [
    top_left_tf(2,1) top_left_tf(1,2);
    top_left_tf(1,1) top_left_tf(2,2);
];

[ min_to_a, ind_to_a ] = min(paths_to_a);

bin_to_a = bins_to_a(ind_to_a,:);

[ min_to_b, ind_to_b ] = min(paths_to_b);

bin_to_b = bins_to_b(ind_to_b,:);


% bottom left
bottom_left = grph(3:4,1:2);

bottom_left_tf = tfs(3:4,1:2);

paths_to_c = [
    bottom_left(1,1) + bottom_left(1,2);
    bottom_left(2,1) + bottom_left(2,2);
];

paths_to_d = [
    bottom_left(2,1) + bottom_left(1,2);
    bottom_left(1,1) + bottom_left(2,2);
];

bins_to_c = [
    bottom_left_tf(1,1) bottom_left_tf(1,2);
    bottom_left_tf(2,1) bottom_left_tf(2,2);
];

bins_to_d = [
    bottom_left_tf(2,1) bottom_left_tf(1,2);
    bottom_left_tf(1,1) bottom_left_tf(2,2);
];

[ min_to_c, ind_to_c ] = min(paths_to_c);

bin_to_c = bins_to_c(ind_to_c,:);

[ min_to_d, ind_to_d ] = min(paths_to_d);

```

```
bin_to_d = bins_to_d(ind_to_d,:);


% top right
top_right = grph(1:2,3:4);

top_right_tf = tfs(1:2,3:4);

paths_from_a = [
    top_right(1,1) + top_right(1,2);
    top_right(2,1) + top_right(2,2);
];

paths_from_b = [
    top_right(1,1) + top_right(2,2);
    top_right(2,1) + top_right(1,2);
];

bins_from_a = [
    top_right_tf(1,1) top_right_tf(1,2);
    top_right_tf(2,1) top_right_tf(2,2);
];

bins_from_b = [
    top_right_tf(1,1) top_right_tf(2,2);
    top_right_tf(2,1) top_right_tf(1,2);
];

[ min_from_a, ind_from_a ] = min(paths_from_a);

bin_from_a = bins_from_a(ind_from_a,:);

[ min_from_b, ind_from_b ] = min(paths_from_b);

bin_from_b = bins_from_b(ind_from_b,:);


% bottom right
bottom_right = grph(3:4,3:4);

bottom_right_tf = tfs(3:4,3:4);

paths_from_c = [
    bottom_right(1,1) + bottom_right(1,2);
    bottom_right(2,1) + bottom_right(2,2);
];

paths_from_d = [
    bottom_right(1,1) + bottom_right(2,2);
    bottom_right(2,1) + bottom_right(1,2);
];

bins_from_c = [
    bottom_right_tf(1,1) bottom_right_tf(1,2);
    bottom_right_tf(2,1) bottom_right_tf(2,2);
];

```

```
bins_from_d = [
    bottom_right_tf(1,1) bottom_right_tf(2,2);
    bottom_right_tf(2,1) bottom_right_tf(1,2);
];

[ min_from_c, ind_from_c ] = min(paths_from_c);

bin_from_c = bins_from_c(ind_from_c,:);

[ min_from_d, ind_from_d ] = min(paths_from_d);

bin_from_d = bins_from_d(ind_from_d,:);


% Second level of reduction
% We have to take the reduced paths to and from each node
% and sum up their weights. Then we need to do simple comparisons
% to determine which path is the shortest.
path_lengths = [
    min_to_a + min_from_a;
    min_to_b + min_from_b;
    min_to_c + min_from_c;
    min_to_d + min_from_d;
];

path_bins = [
    horzcat(bin_to_a,bin_from_a);
    horzcat(bin_to_b,bin_from_b);
    horzcat(bin_to_c,bin_from_c);
    horzcat(bin_to_d,bin_from_d);
];

[ shortest_path_length, index ] = min(path_lengths);

% shortest path binary
s_p_bin = de2bi(sum(path_bins(index,:)),8);

shortest_path_bin = [ s_p_bin(1:3) s_p_bin(5) s_p_bin(4) s_p_bin(6:8) ];
```

2.4 Bit Error Rate with Gaussian Channel

Listing 3: Bit Error Rate vs. SNR

```
results = zeros(1,10);
message_bits = zeros(1,10);

for i = 1:10
    x = generate_binary_values(4);
    for j = 1:1000
        ind = ceil(rand() * 16);
        result = two_c_one(awgn(encoder_844(x(ind,:)),i));
        num_errs = sum(xor(x(ind,:), result(1:4)));
        results(i) = results(i) + num_errs;
        message_bits(i) = message_bits(i) + 4;

```

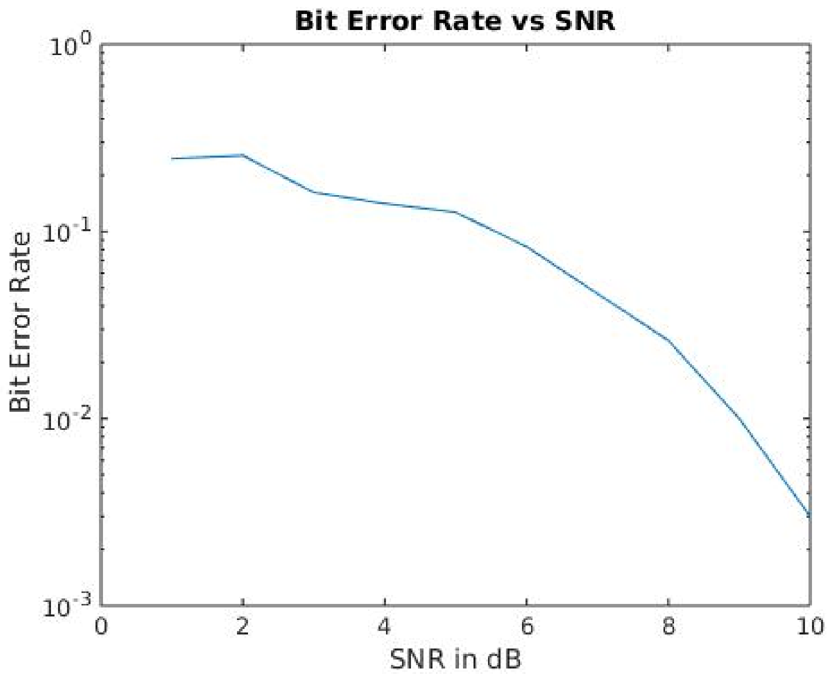


Figure 2: Bit Error Rate vs. SNR of Gaussian Channel

```
        if results(i) > 100 || message_bits(i) > 10000
            break;
        end
    end
end


bER = results ./ message_bits;

figure
semilogy(1:10,bER)
title('Bit Error Rate vs SNR')
xlabel('SNR in dB')
ylabel('Bit Error Rate')
```

3 Question 3

(a) In this plot, the first non-zero value is index 149 which corresponds to a distance of 3.1960 meters when using 343.2 m/s as the speed of sound in the air.

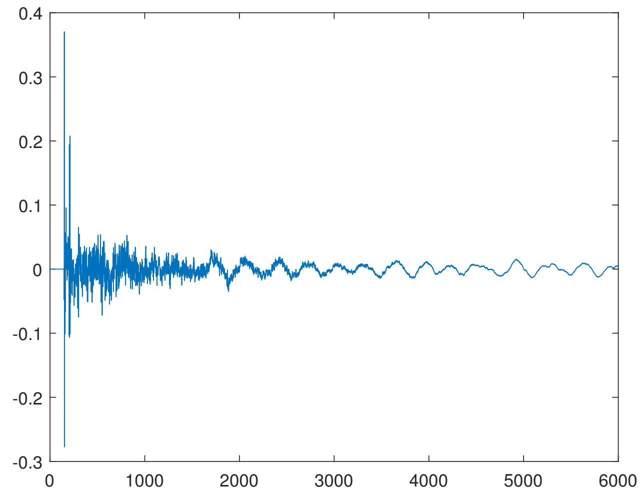


Figure 3: Plot of the impulse response

Listing 4: Compute distance to microphone

```
plot(h)
i = 1;
while (true)
    if h(i) ~= 0
        i;
        break;
    else
        i = i + 1;
    end
end

% assuming speed of sound is 343.2 m/s in dry air
distance = i / 16000.0 * 343.2
```

(b) We can compute the convolution to get the system response to the speech file using a built-in matlab function:

This operation adds a reverb effect to the speech track. Instead of a very dry and flat voice, we get the impression it was recorded in a big hall.

Listing 5: Convolution function

```
out = conv(x,h);
sound(out, 16000)
```

(c) This new impulse response can be generated in the following way:

Listing 6: Convolution function

```
function [ echo ] = delay( x, delay )
echo = [x; zeros(delay,1)] + [zeros(delay,1); x];
end

out = conv(x, delay(h, 3000));
sound(out, 16000)
```

This operation adds an echo to the speech file. We hear the same voice, with the reverb, just echoed.

(d) To play the signal backward, you simply need to flip the array around, in Matlab this looks like the following:

Listing 7: Time inversion method

```
out = flip(input);
sound(out, 16000)
```

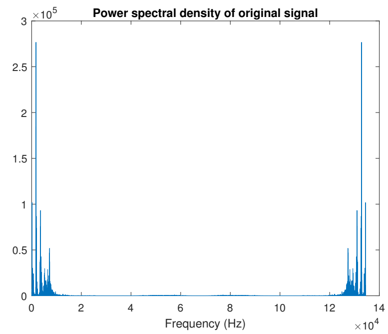
.sdrawkcab langis eht raeh eW
We hear the signal backwards.

(e) Changing the playback speed change the pitch of the voice. The speeds tested don't really impact the clarity of the speech, the text is easily understood, but the voice's pitch is higher when played at a faster speed, and inversely.

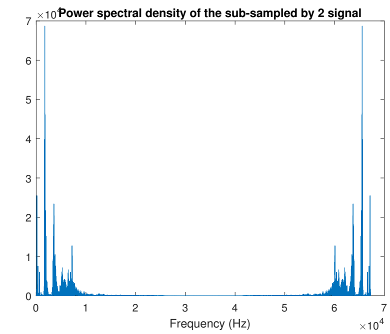
(f) By sub-sampling the signal, we loose higher frequency information. Upon playback, we can hear the distortion, the voice is much lower (in frequency) and all the high frequencies are absent.

(g) The process of quantization changes the precision with which the amplitude of the signal is represented. As such, with a lower number of quantization bits, the signal we heard was very distorted and the voice cracked a lot more. These effects were reduced as we increase the number of bits.

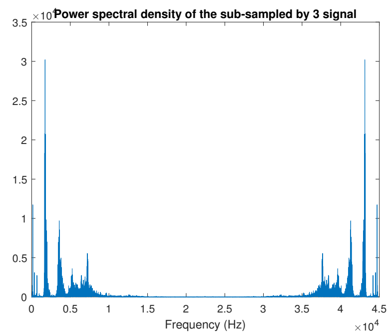
To achieve the quantization, we designed an algorithm that scales all the



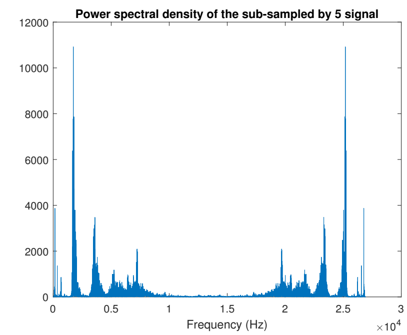
(a) Original power spectrum



(b) Power spectrum with 2:1 sub-sampling



(c) Power spectrum with 3:1 sub-sampling



(d) Power spectrum with 5:1 sub-sampling

Figure 4: Comparison of power spectrum when sub-sampling

samples in a range of 0 to 1, then scales them back to the maximum range allowed by the number of quantization bits. Finally we take the integer approximation of that number to get quantized data.

Listing 8: Quantization algorithm

```
% bring signal in [0:1] range
normalized_signal = abs(signal)/max_value;

% Multiply by quantization levels and floor, this is the
    quantizatino step
quantized_signal = int64(normalized_signal * (no_of_levels
    -1));
```

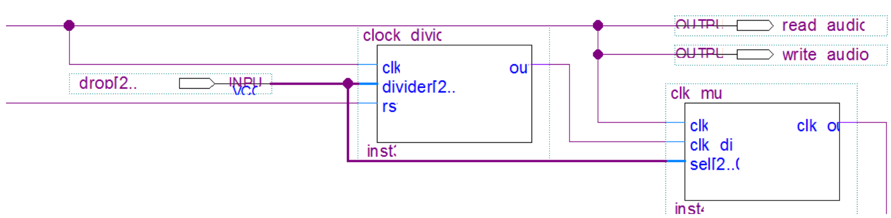
4 Question 4

(a) To quantize the real-time signal, we considered only the most significant bits, and quantized the signal by keeping only those, and setting the rest to zero, as shown bellow.

Listing 9: Quantization algorithm

```
module quantizer (level, in, clk, out);
        input [2:0] level;
        input [31:0] in;
        output [31:0] out;
        input clk;

        reg [31:0] filter;
        always@(posedge(clk))
        begin
        case (level)
            3'b111: filter = 32'b11111111000000000000000000000000;
            3'b100: filter = 32'b11110000000000000000000000000000;
            3'b011: filter = 32'b11100000000000000000000000000000;
            3'b010: filter = 32'b11000000000000000000000000000000;
            3'b001: filter = 32'b10000000000000000000000000000000;
            3'b000: filter = 32'b11111111111111111111111111111111;
                endcase
                out = filter & in;
        end
endmodule
```

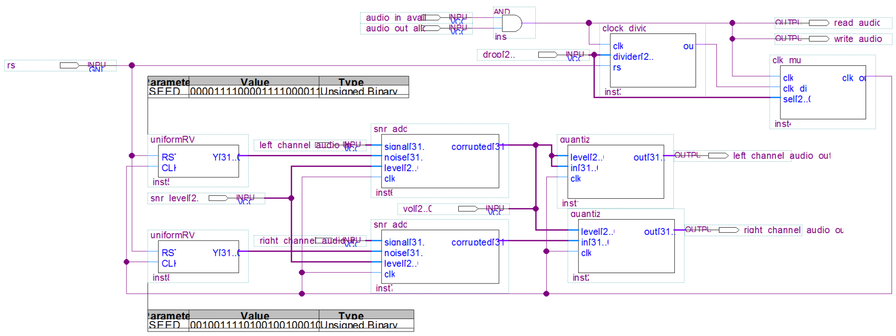
(b) We implemented subsampling using a clock divider. The circuit below demonstrates:



In this circuit, two verilog modules are used to create a rescaled clock (listing 15), then one to select between the scaled and original clock (listing 16).

(g) To add white noise to the signal, we used the uniform random number generator provided and added that number to the number representation of the sample. To change the SNR level, we simply shifted the noise value by a certain number of bits. To simplify the taks, we used the approximation that the 4th most significant bit represented the 0dB level, and that a 1 bit shift represented a 6dB change. Based on those approximation, we generated the variable added module found in listing 17.

Our full filtering cuicuit is the following:



5 Signal Processing for Binary Erasures

5.1 Real vs. Mod 2

In the equation $Ax = b$ A $= \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ and b $= \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$

x in the mod 2 domain is restricted to being $\begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$ while in the real domain it could be this but it could also be $\begin{pmatrix} 2 \\ -1 \\ 1 \\ 1 \end{pmatrix}$.

5.2 Binary vs. Mod 2

A $= \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$ b $= \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$ yields x $= \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$ in both the binary and mod 2 domain while $\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$ is also a valid result in the mod 2 domain.

The reason for this is we have a column in the matrix A with only one 1 which coincides with an outlier value in the result. This pair with the fact that we have one other 1 in this same row in the A, while the rest of the matrix (i.e. the 3x3 in the top left) corresponds to the permutations of two 1's. In the binary state, the a_{43} forces the a_{44} to 1 while in the mod 2 domain there is no such restriction.

5.3 Conclusion for Implementation

We need to handle the fact that there can be multiple correct solutions in the mod 2 arithmetic case. We need to anticipate the fact that sometimes there will be an unresolvable state. We handle this by simply guessing 1 for the first unknown.

5.4 Gaussian Elimination MatLab

Listing 10: Gaussian Eliminator

```
function [ solution ] = five_d(coef, res)
% Solves the 4 equation with 4 unknown system of equations represented by
% coef by comparing against res and using mod-2 arithmetic.

% We can not give the result for this if we do not have independent coefs.
if(rank(coef) == 4)
    solution = zeros(4,1);
    solved_positions = zeros(4,1);

    while sum(solved_positions)<4
        change = 0;
        for i = 1:4
            %Only can actually properly decide if there is exactly one unknown
            current_row = coef(i,:);
            number_of_unknowns = current_row * ~solved_positions;
            if number_of_unknowns == 1
                change = 1;
                index = find(and(current_row, ~solved_positions'),1);
                solved_indeces = find(solved_positions);
                cur_row_solved = current_row(solved_indeces);
                sols = solution(solved_indeces);
                solution(index) = mod(sum(and(cur_row_solved, sols'))+res(i),2);
                solved_positions(index) = 1;
            end
        end
        %If we have to, we just set the first unsolved spot to 1.
        if ~change
            ind = find(~solved_positions,1);
            solution(ind) = 1;
            solved_positions(ind) = 1;
        end
    end
end
```

6 Question 6

(a) If a LTI system is non-causal, then its impulse response $h[n]$ must be non-zero for some $n < 0$. Let's take an impulse response which is non-zero for some $n < 0$: $h[n] = \delta[n+1]$

We can find the output equation of the system:

$$\begin{aligned} y[n] &= x[n] * h[n] \\ FT(y[n]) &= FT(x[n])FT(h[n]) \\ Y(e^{j\omega}) &= X(e^{j\omega})e^{j\omega 1} \\ y[n] &= x[n+1] \end{aligned}$$

Clearly, this system is non-causal, showing that an impulse response $h[n] \neq 0$ for $n < 0$ leads to a non-causal system.

(b) To show that the absolute summation of the impulse response is a sufficient test for BIBO stability, let's consider an impulse response that is not absolutely summable: $\sum_{k=-\infty}^{\infty} |h[k]| = \infty$

Now let's consider a bounded input function $x[n] = \text{sign}(h[-n])$:

$$x[n] = \begin{cases} 1 & \text{if } h[-n] \geq 0 \\ -1 & \text{if } h[-n] < 0 \end{cases}$$

Next, let's compute the convolution sum for this system:

$$\begin{aligned} y[n] &= x[n] * h[n] \\ &= \sum_{k=-\infty}^{\infty} \text{sign}(h[-k])h[n-k] \\ y[0] &= \text{sign}(h[-0])h[-0] + \text{sign}(h[-1])h[-1] + \text{sign}(h[1])h[1] + \text{sign}(h[-2])h[-2] + \cdots \\ &= |h[n]| = \infty \end{aligned}$$

If the output is un-bounded for at least a single input value, the test fails, and the system is not BIBO stable, as is the case here.

(c) The echo system in Question 1 can be described by the following equation:

$$y[n] = x[n] + x[n-160] + x[n-1600] + x[n-3200] + x[n-8000]$$

And $h[n] = \delta[n] + \delta[n-160] + \delta[n-1600] + \delta[n-3200] + \delta[n-8000]$ To find the inverse system, we take the Fourier transform and its reciprocal:

$$\begin{aligned} H(e^{j\omega}) &= DTFT(\delta[n] + \delta[n-160] + \delta[n-1600] + \delta[n-3200] + \delta[n-8000]) \\ &= 1 + e^{-160j\omega} + e^{-1600j\omega} + e^{-3200j\omega} + e^{-8000j\omega} \end{aligned}$$

Using Matlab, we can plot that frequency response, and clearly see that there are frequencies that make this function 0, so it is not invertible.

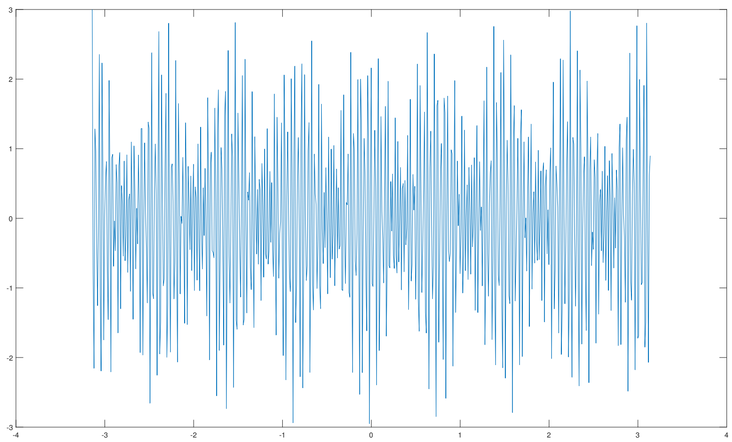


Figure 5: Frequency response of the echo system

(d) We can still use the z-transform to find a pseudo-inverse system:

$$\begin{aligned} H(z) &= z^{-160} + z^{-1600} + z^{-3200} + z^{-8000} \\ H(z)^{-1} &= \frac{1}{z^{-160} + z^{-1600} + z^{-3200} + z^{-8000}} \end{aligned}$$

And we can create a valid block diagram for that system:

(e) The convolution theorem states that the response of a system $y[n]$ to input signal $x[n]$ is determined by the convolution operation with the system's impulse response $h[n]$. To prove this theorem, we use two properties: linearity and time invariance.

The former states that if $x1$ results in $y1$ and $x2$ results in $y2$ that $x1+x2$ will result in $y1+y2$, and the lather states that it $x[n]$ results in $y[n]$, that shifting the input in time, simply shifts the output by the same amount, or $x[n-\Delta]$ results in $y[n-\Delta]$.

Based on these results, we can consider our input function as a combination of time-shifted impulse functions:

$$x[n] = x[0]\delta[n] + x[1]\delta[n-1] + x[-1]\delta[n+1] + \cdots$$

If we consider the individual samples going through the system:

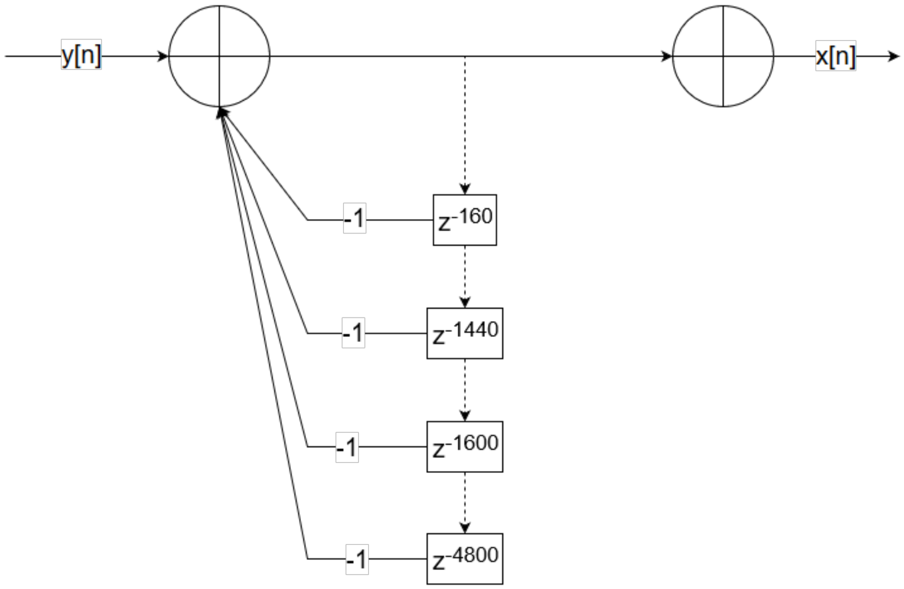


Figure 6: Echo removal system block diagram

The key observation here is that we can use linearity to sum the input and output signals. Specifically, if we sum the input, we get back our

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

If we sum the output, we get

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

This can be rewritten as

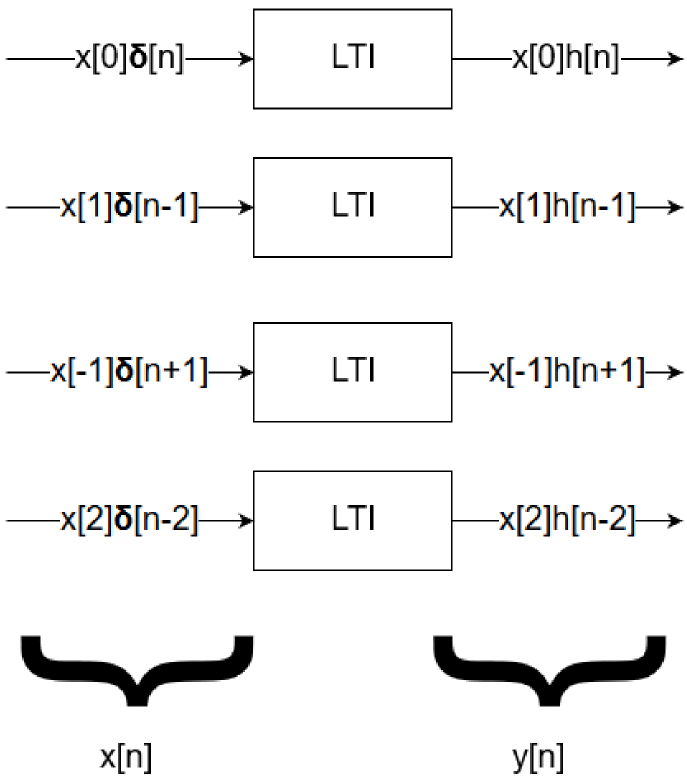
$$\sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

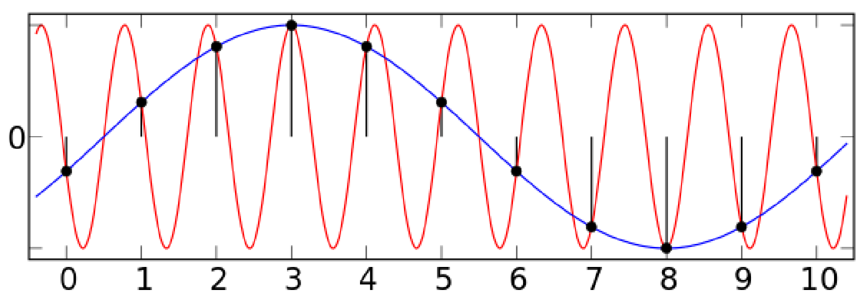
which is the definition of the convolution operation. This shows how $y[n] = x[n] * h[n]$.

(f) Will do later.

(g) Aliasing is a phenomena in which two (or more) continuous functions could map to a single set of discrete samples. A great illustration of this can be found on wikipedia:

In this picture, we can see that a high frequency signal and a low frequency signal would match the set of samples. This is because the high frequency signal's frequency is larger that half the sampling frequency.





This contradicts the Nyquist theorem, which states that in order to properly reconstruct a signal, it must be sampled at a frequency at least double of the sampled signal's frequency.

In this case, this requirement is not respected, which leads to this aliasing. In fact, anytime a signal is sampled without respecting the Nyquist criterion, it will lead to aliasing, and to won't be possible to reconstruct it properly.

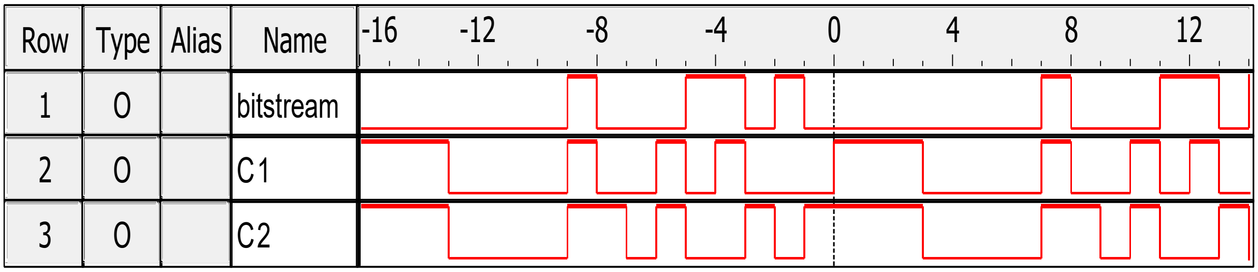
7 Appendices

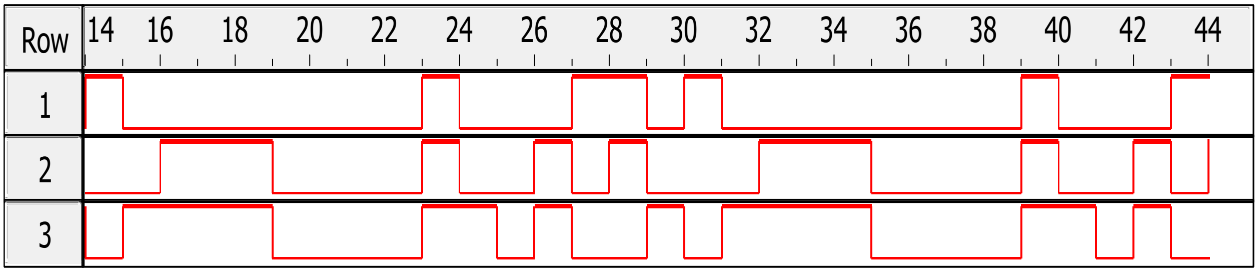
Listing 11: generate_binary_values

```
function [ output] = generate_binary_values( bits )
    % Generate all possible binary string of some width
    %   Arguments:
    %       bits: width in bits of the string

    output = zeros(2^bits, bits);

    value = 0;
    for col = 1:bits
        row = 0;
        for rep = 1:2^(bits-col+1)
            for item = 1: 2^(col-1)
                row = row + 1;
                output(row, bits-col+1) = value;
            end
            value=mod (value + 1, 2);
        end
    end
end
```

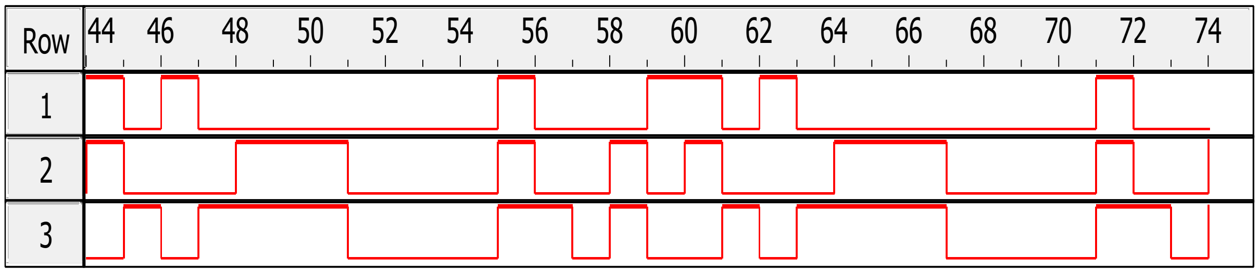
Listing 12: encoder_844

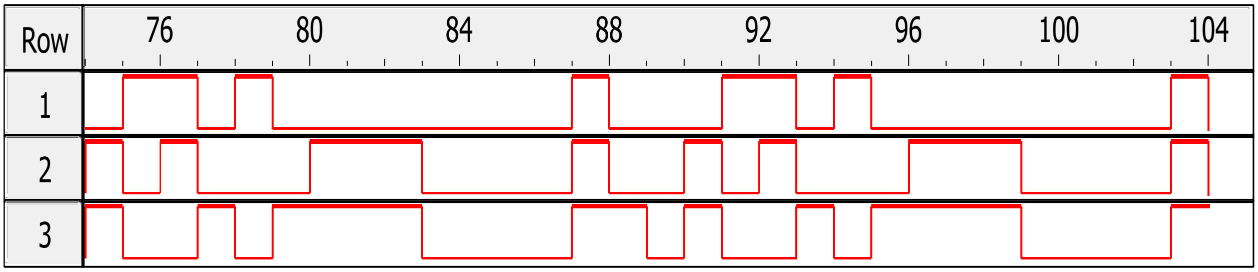
```
function [ c ] = encoder_844( m )
    % Encodes the 4 bit message m to the 8 bit codeword based on
        the (8,4,4)
    % encoding scheme.

    % Get the parities
    p = [];
    p(1) = mod(m(1) + m(2) + m(3), 2);
    p(2) = mod(m(1) + m(2) + m(4), 2);
    p(3) = mod(m(1) + m(3) + m(4), 2);
    p(4) = mod(m(2) + m(3) + m(4), 2);

    c = horzcat(m, p);
end
```

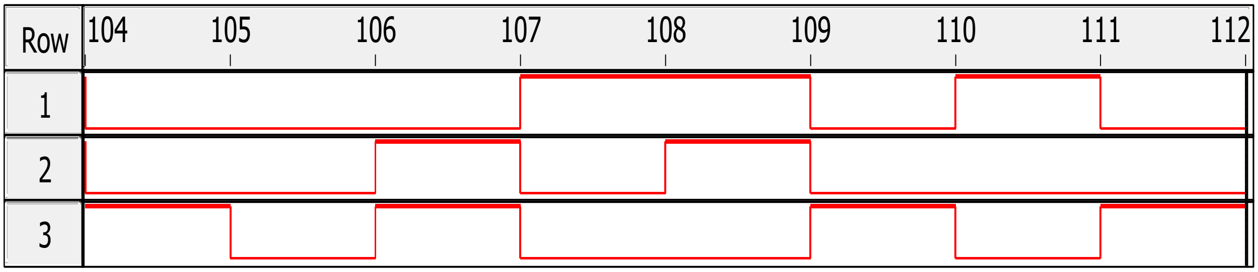

Row
Type
Alias
Name
-16
-12
-8
-4
0
4
8
12
1
O
bitstream
2
O
C1
3
O
C2



Row
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
1
2
3


Row
44
46
48
50
52
54
56
58
60
62
64
66
68
70
72
74
1
2
3


Row
76
80
84
88
92
96
100
104
1
2
3


Row
104
105
106
107
108
109
110
111
112
1
2
3

Listing 13: Echo verilog circuit generation script

```python

#!/usr/bin/env python
import sys

if __name__ == "__main__":
    try:
        reg_count = int(sys.argv[1])
        filename = "reg" + str(reg_count) + ".v"

    except IndexError as e:
        print(e)
        sys.exit(255)

    except ValueError as e:
        print("Lenght is not an integer")
        sys.exit(1)

    fileoutput = open(filename, 'w')
    fileoutput.write("module reg" + str(reg_count) + " ( in, clk, rst, out );\n")
    fileoutput.write("\tinput[31:0] in;\n")
    fileoutput.write("\tinput clk;\n")
    fileoutput.write("\tinput rst;\n")
    fileoutput.write("\toutput[31:0] out;\n\n")

    for i in range(0, reg_count):
        fileoutput.write("\twire [31:0] r" + str(i) + "_out;\n")

    fileoutput.write("\n")
    for i in range(0, reg_count):
        if i == 0:
            in_sig = "in"
        else:
            in_sig = "r" + str(i-1) + "_out"

        fileoutput.write("\treg32 r" + str(i) + " (rst, clk, " + in_sig + ", r" + str(i) + "_out);\n")

    fileoutput.write("\n")
    fileoutput.write("\tassign out = r" + str(reg_count-1) + "_out;\n")
    fileoutput.write("endmodule\n")

    fileoutput.close()
```

Listing 14: Verilog code for echo

```
module reg10 ( in, clk, rst, out );
        input[31:0] in;
        input clk;
        input rst;
        output[31:0] out;

        wire [31:0] r0_out;
        wire [31:0] r1_out;
        wire [31:0] r2_out;
        wire [31:0] r3_out;
        wire [31:0] r4_out;
        wire [31:0] r5_out;
        wire [31:0] r6_out;
        wire [31:0] r7_out;
        wire [31:0] r8_out;
        wire [31:0] r9_out;

        reg32 r0 (rst, clk, in, r0_out);
        reg32 r1 (rst, clk, r0_out, r1_out);
        reg32 r2 (rst, clk, r1_out, r2_out);
        reg32 r3 (rst, clk, r2_out, r3_out);
        reg32 r4 (rst, clk, r3_out, r4_out);
        reg32 r5 (rst, clk, r4_out, r5_out);
        reg32 r6 (rst, clk, r5_out, r6_out);
        reg32 r7 (rst, clk, r6_out, r7_out);
        reg32 r8 (rst, clk, r7_out, r8_out);
        reg32 r9 (rst, clk, r8_out, r9_out);

        assign out = r9_out;
endmodule
```

Listing 15: Clock divider module

```
module clock_divider (clk, divider, rst, out);

	input clk;
	input [2:0] divider;
	input rst;
	output out;

	integer count;
	reg output_reg;
	reg [2:0] last_div;



always@(posedge clk)
begin
	if (rst)
	begin
		output_reg = 0;
		count = 0;
		last_div = 0;
	end
	else
	begin
		if (last_div == divider)
			count = (count + 1);
		else
			count = 0;

		if (count == divider)
		begin
			output_reg = ~output_reg;
			count = 0;
		end
	last_div = divider;
	end
end
assign out = output_reg;

endmodule
```

Listing 16: Clock selection module

```
module clk_mux (clk, clk_div, sel, clk_out);

    input clk;
    input clk_div;
    input [2:0] sel;
    output clk_out;

    assign clk_out = (sel == 3'b000) ? clk : clk_div;

endmodule
```

Listing 17: Noise adder module

```
module snr_adder (signal, noise, level, clk, corrupted);
    input [31:0] signal;
    input [31:0] noise;
    output [31:0] corrupted;
    input [2:0] level;
    input clk;

    always @(posedge clk) begin
        if (level == 3'b111)
            // SNR = -10dB
            corrupted = signal + (noise >>> 2);
        else if (level == 3'b000)
            // SNR = 0dB
            corrupted = signal + (noise >>> 4);

        else if (level == 3'b001)
            // SNR = 10dB
            corrupted = signal + (noise >>> 6);
        else if (level == 3'b010)
            // SNR = 20dB
            corrupted = signal + (noise >>> 7);
        else if (level == 3'b011)
            // SNR = 30dB
            corrupted = signal + (noise >>> 9);
        else if (level == 3'b100)
            // SNR = 40dB
            corrupted = signal + (noise >>> 11);

    end
endmodule
```