Brian Dallaire, Erik Michel
EE 371
5/23/2021

Lab 5 Report

Procedure

This lab consists of three tasks where task 2 and 3 are the main tasks. The first is to perform a simple recording test using a *piano.mp3* file and playing the file into the FPGA. Additionally, the first task is used to test if the sound output of the FPGA can be properly altered. The second task is to implement a filter that will remove the high-pitched noise we introduced into our audio output in task 1 by designing a Finite Impulse Response (FIR) filter. The third task is to improve on the task 2 design by making a generalized FIR filter for any buffer of size N. Furthermore, the task 3 design will use additional components compared to task 2 in order to achieve a better filtering result.

Task 1

Procedure

The first task, to perform a recording test using a *piano.mp3* file on the FPGA, was performed by using LabLands, the provided SystemVerilog files, and *piano.mp3* to perform the recording where we began with CASE A, from the lab specification, to record the initial 10 seconds and ended with CASE B for the remaining 10 seconds of a 20 second total recording. Before synthesizing these components, a testbench was created for the noise_gen.sv module.

Results

The result was a file named *task1.mp3* (included in the demo portion of this lab) that contained the audio as described above and specified by CASE A and CASE B from the lab specification. On our end this audio could be described as *piano.mp3* playing undistorted for the initial 10 seconds and ending with a layer of audible noise for the remaining 10 seconds.

Figure 1 shows the ModelSim waveform produced by the noise_gen_testbench module from noise_gen.sv (appx. Universal.1).

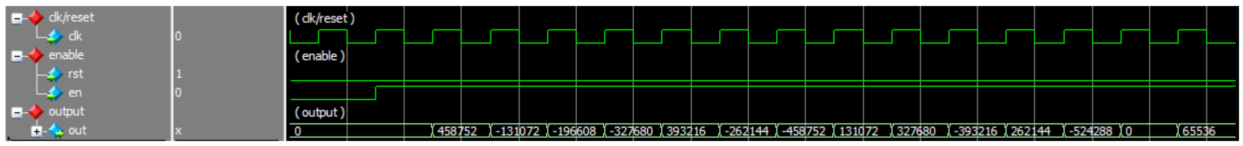


Figure 1: ModelSim waveform created by noise_gen testbench

Block Diagram - Task 1

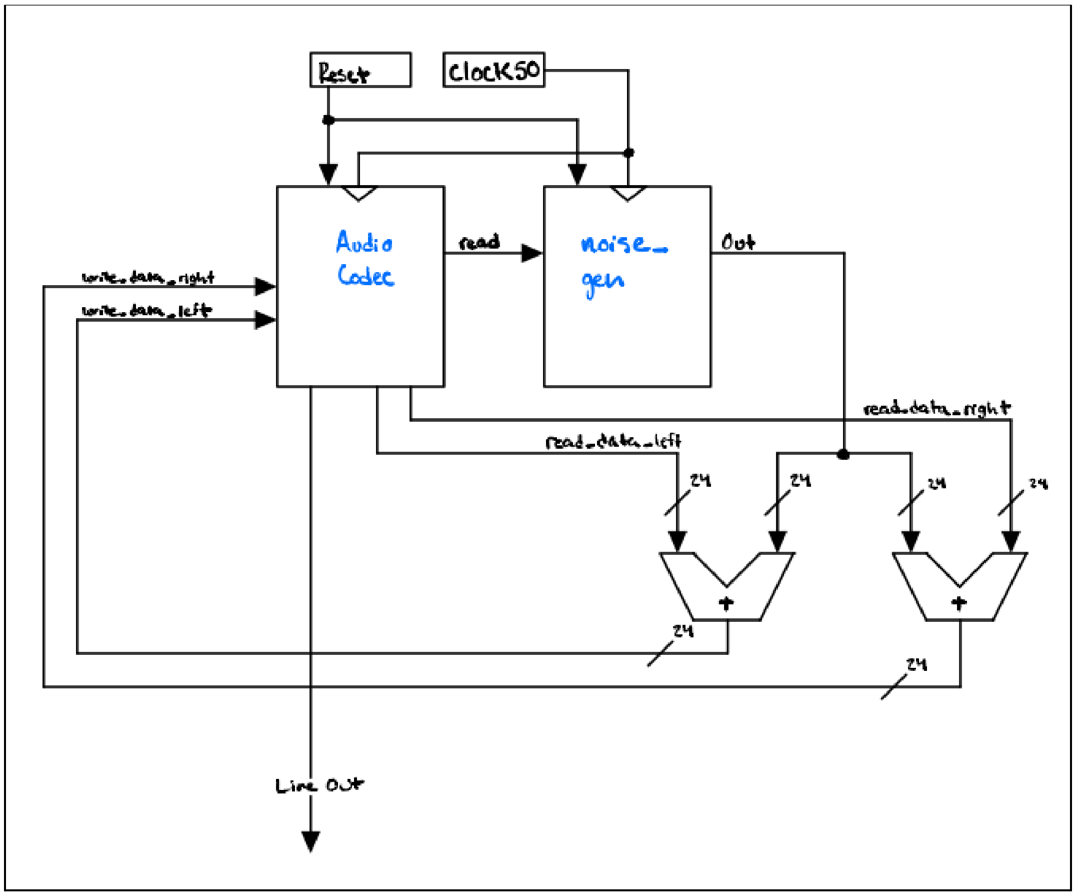


Figure 2: Block diagram of the task 1 circuit

Design Decisions

Task 1 did not require any major design decisions as the materials and steps to complete this task were given in advance and did not require any additional design process or implementation.

Task 2

Procedure

The second task was to implement a simple averaging FIR filter to mitigate the high frequency sound introduced in Task 1. An averaging filter can remove noise from a sound by averaging values from multiple samples. As shown in Figure 3, in this task we removed small deviations in sound by averaging the 8 adjacent samples.

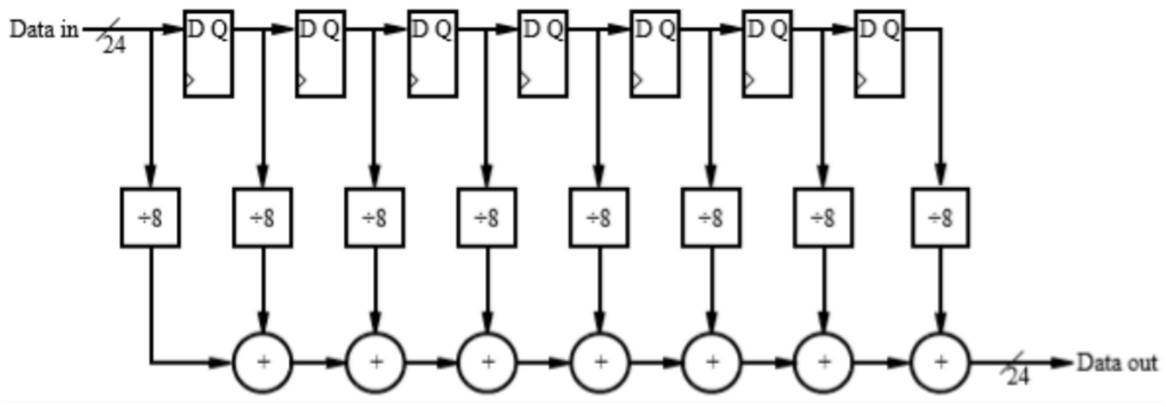


Figure 3: Circuit diagram of the task 2 circuit

In essence, the circuit is dividing the input data by 8 in 8 different instances, separated by 7 registers. As long as the read input is true, each register will move the input data to the next register every clock cycle. The output of each register is divided by 8 and added towards data out. By passing the input data sound through this filter, it will supposedly remove small deviations from the noisy audio and lessen the high frequency noise. However, at 48 kHz, averaging the 8 adjacent samples is a very small time frame, and this will not be the most effective filter. That being said, the results should still be noticeable.

Results
The result of task 2 was successfully implementing a small averaging FIR filter that averages the 8 adjacent samples. Shown below in Figure 4 is the ModelSim Simulation for FIR_filter.sv, which showed that our module successfully accomplishes the behavior described in the specifications.

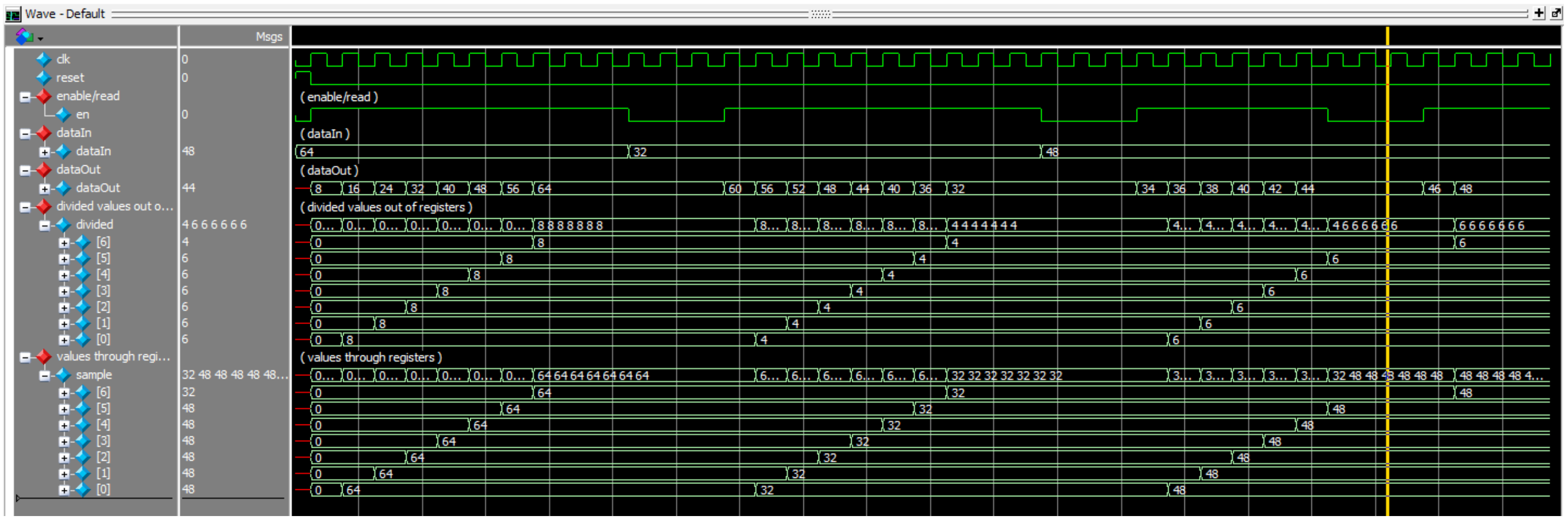


Figure 4: ModelSim waveform for FIR_filter.sv

In this simulation, we tested if the input data will be averaged out properly, and if new samples would not be accepted when read is false. Alternatively, we also tested what happens when read is false before every register holds the value of data in.

Design Decisions
Originally, the approach towards this task was simple. Create seven registers and divide each of their outputs and add them together along with the initial divided value to determine data out. This method worked perfectly fine and was potentially the expected solution for this task. However, my partner and I decided that making this module using a generate statement will better prepare us for task 3, which is a filter with variable samples. A generate statement allowed us to only call the register module twice rather than 7 times from our old method and accomplish the same result. Because our generate statement used a for loop, the number of averaging samples could be variable. Although Task 2 and Task 3 work differently, our experience using the generate statement for Task 2 helped us prepare for the Task 3 solution we designed. Our solution involving the generate statement was over complicated and overkill for this task, but it produced the equivalent, desired result as our old method (with less code) so we stuck with it.

Block Diagram - Task 2

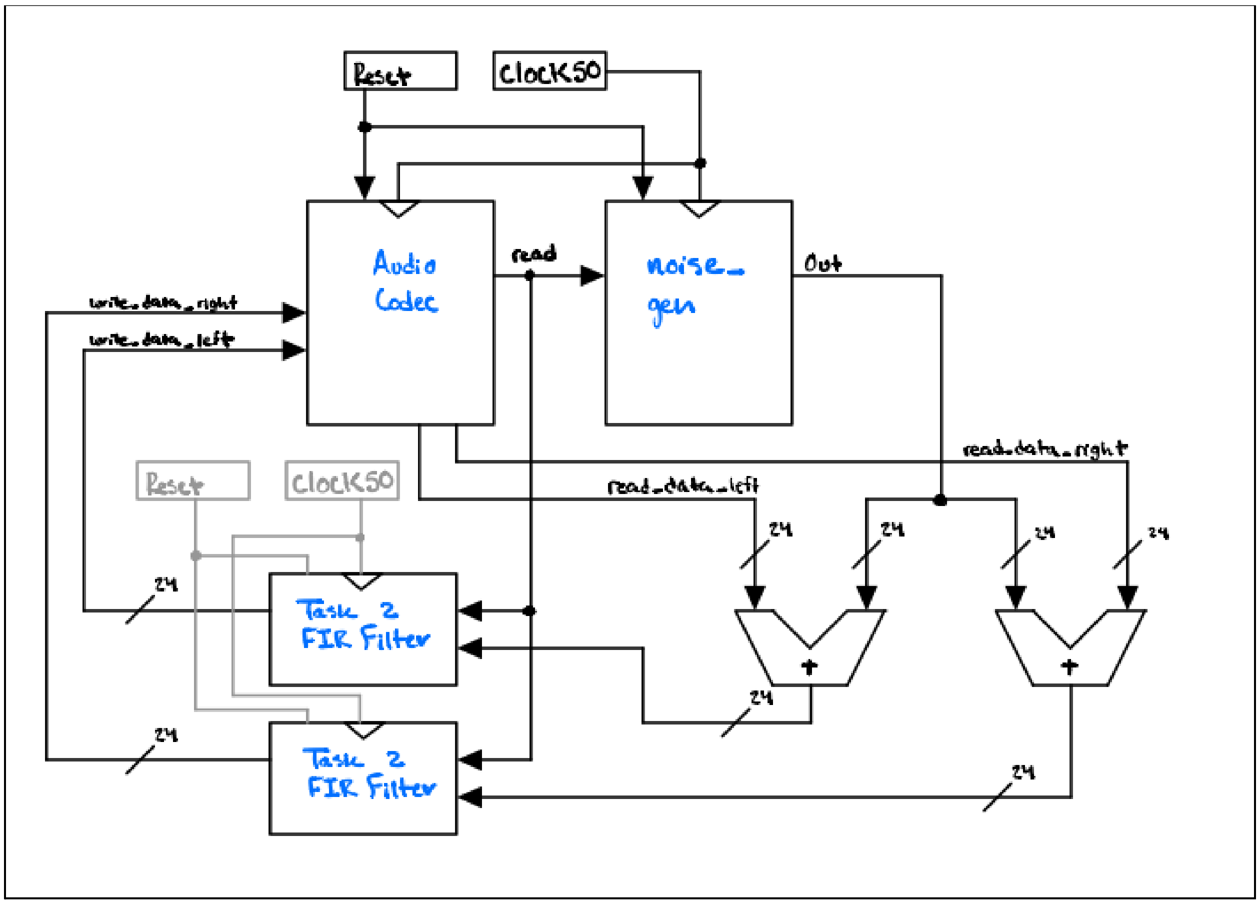


Figure 5: Block diagram of the task 2 circuit

Task 3

Procedure

The third task was to create a generalized averaging filter that can take a N number of samples to average. This filter varies a little bit from the filter shown in Task 2. Unlike Task 2, there is only one division after inputting the data in. This divided value will go into a buffer of size N, and the oldest value out of this buffer will be subtracted from the next divided value of data in. Additionally, in task 3 there is the addition of an accumulator. The accumulator is a register that will input the current value of data out and output it the next clock cycle. The overall value of data out will be the input data divided by N - the value of the oldest value from the N size buffer + the value from the accumulator. The overall circuit is shown in Figure 6.

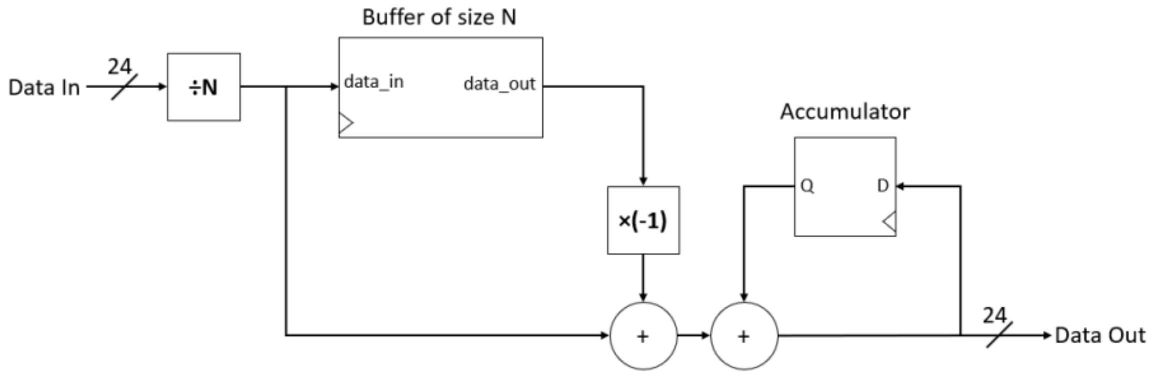
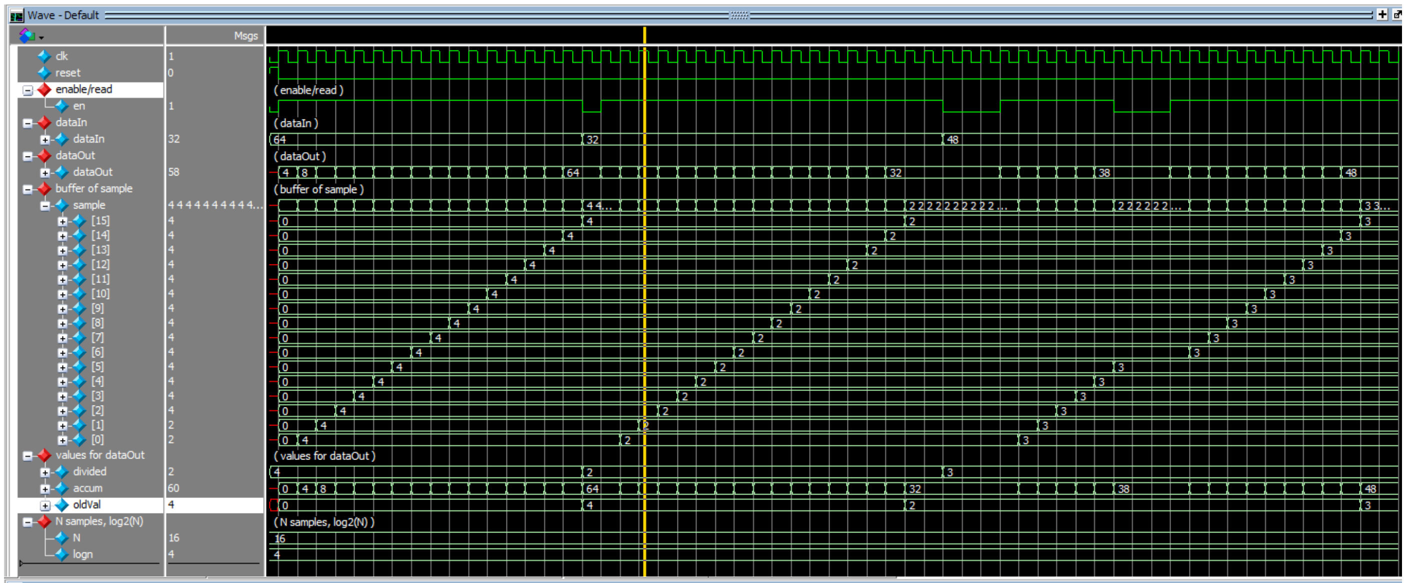


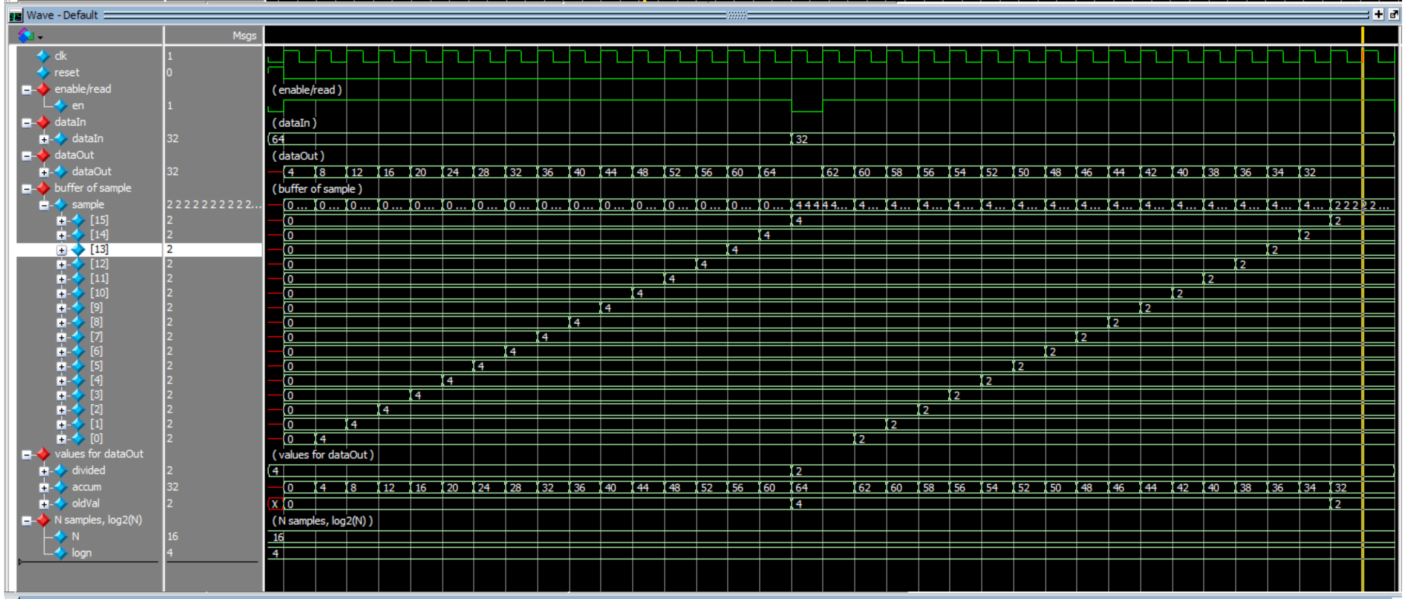
Figure 6: Circuit diagram of the task 3 circuit

In this task we were told to use a N value of 16. This means that the buffer will take 16 clock cycles for the first input to go through and be considered the “oldest value”. The value of N also determines how we divide the incoming data. The value of data in should be divided by 16, and this divided value will go into the buffer as well as the summation for data out. In our code shown in appendix 5, we used the generate statement we practiced from Task 2 for the buffer. Using the generate statement, we successfully made a variable size buffer using only 2 calls for the DFF module. After creating the buffer, it was simple. We assigned data out to be the summation of the input data divided by N, minus the oldest value from the buffer, plus the accumulator. We called the accumulator separate from the generate statement and it had an input of data out and the output was the accumulator value that is added into the summation for data out.

Results

Overall, by using generate statements, we successfully completed Task 3 and the results showed that it was behaving appropriately. We successfully implemented a generalized averaging filter that averages N number of samples. Shown in Figure 7 is our ModelSim Simulation for nSamp_FIR.sv, the module we created that implements the filter for Task 3.


Wave - Default
Msgs
clk
reset
enable/read
en
dataIn
dataIn
dataOut
dataOut
buffer of sample
sample
values for dataOut
divided
accum
oldVal
N samples, log2(N)
N
logn


Wave - Default
Msgs
clk
reset
enable/read
en
dataIn
dataIn
dataOut
dataOut
buffer of sample
sample
values for dataOut
divided
accum
oldVal
N samples, log2(N)
N
logn

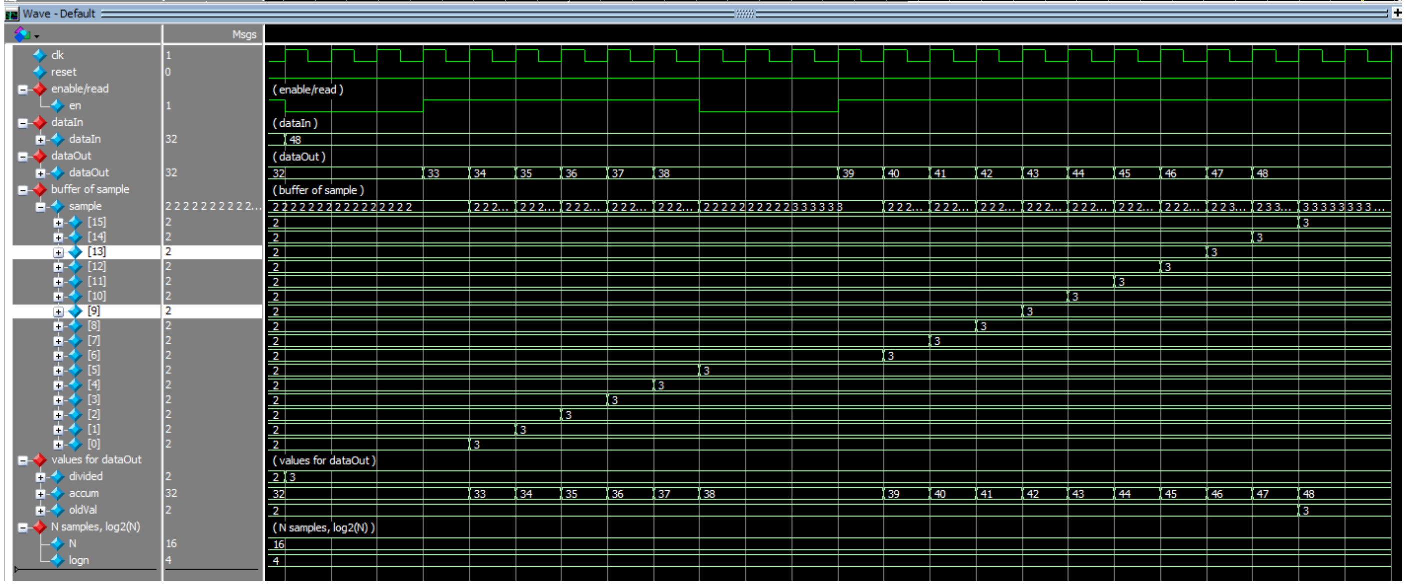

Wave - Default
Msgs
clk
reset
enable/read
en
dataIn
dataIn
dataOut
dataOut
buffer of sample
sample
values for dataOut
divided
accum
oldVal
N samples, log2(N)
N
logn

Figure 7: ModelSim waveform for nSamp_FIR.sv. Top is full waveform, middle is zoom in going from one input value to another, and the bottom is zoom in of making read signal be false before output is complete

In this simulation, we tested if the input data will be averaged out properly, and if new samples would not be accepted when read is false. Alternatively, we also tested what happens when read is false before every register holds the value of data in.

Design Decisions

In this task we took what we learned from our optimized code in Task 2 and implemented it into our Task 3 solution. We learned how to use generate statements to accomplish the desired result with as little code as possible. The generate statement was used to implement the buffer of size N, and because our generate statement uses a for loop, this was as simple as changing the global parameter of the module to be N and making the for loop condition based on N. After the buffer was done, the rest was quite simple. Created a 2D array that represents the buffer. The last value (N-1'th value) from this array was what we considered to be the oldest value from the buffer, so we simply assigned this value to be the oldest. From here, all we have to do is add together the first division of the input data, subtract the oldest value from the buffer, and add the value of the accumulator. Our patience in learning generate statements to resolve Task 2 paid off and designing our Task 3 solution using generate statements was a success.

Block Diagram - Task 3

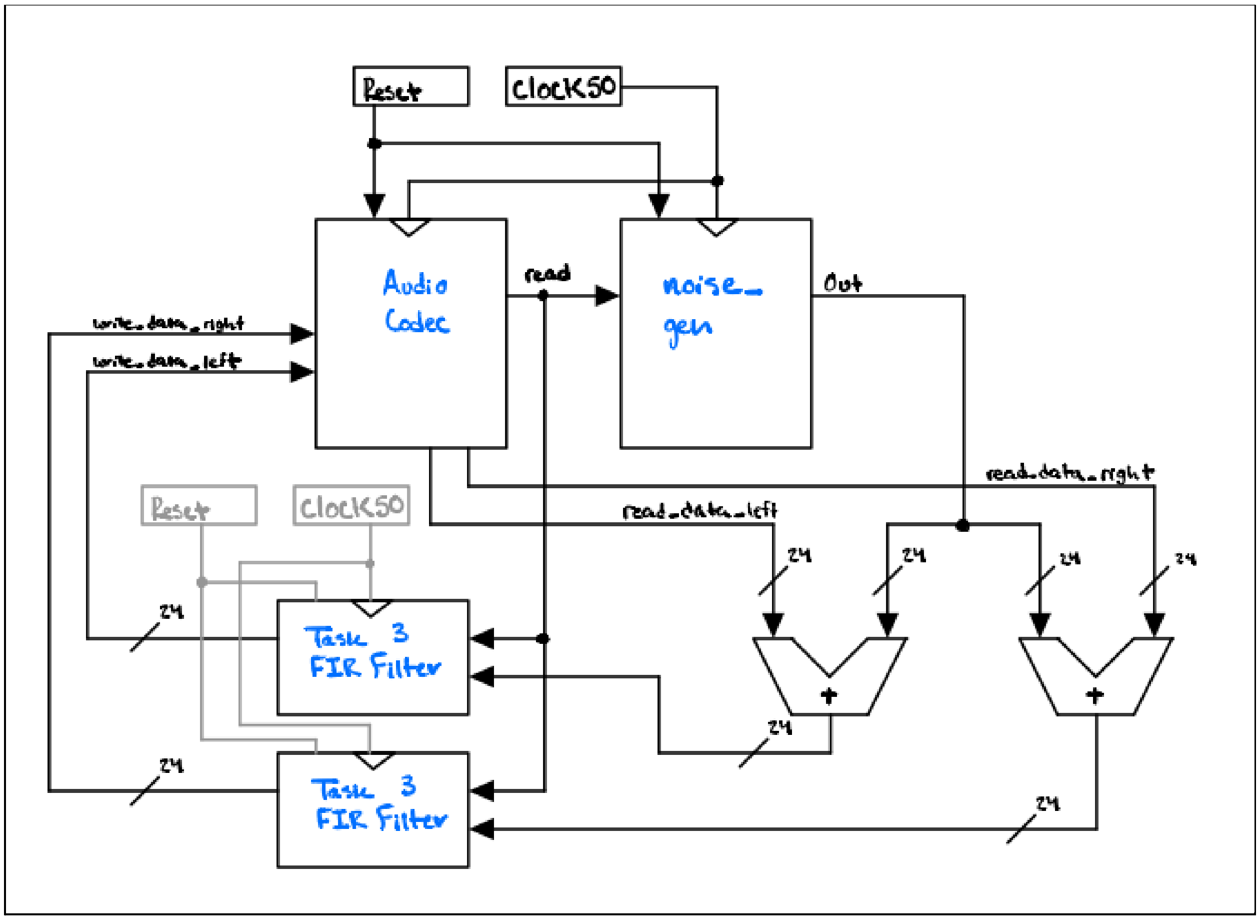


Figure 8: Block diagram of the task 3 circuit

Summary Conclusion

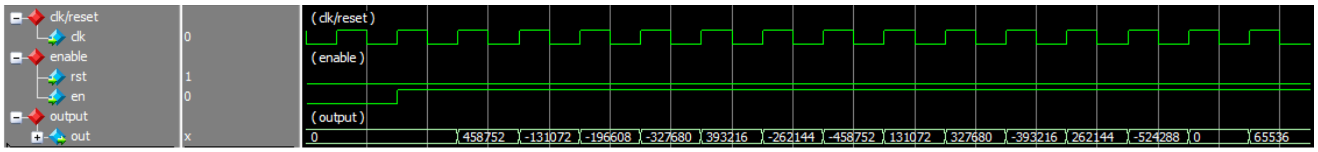
The main goal of this lab was to learn how to manipulate audio files using SystemVerilog code through the FPGA board. Task 1 introduced us to the parts required to output an audio and what an unwanted sound may sound like. Task 2 introduced us to a small, simple averaging FIR filter that could help mitigate noise from sound. Task 3 builds off Task 2 and we had to create an averaging FIR filter with variable size that worked a little differently than Task 2. As we tackled Task 2 and 3, we learned a new way to implement SystemVerilog code by using generate statements. The generate statement was extremely helpful for us to learn to tackle Task 3.

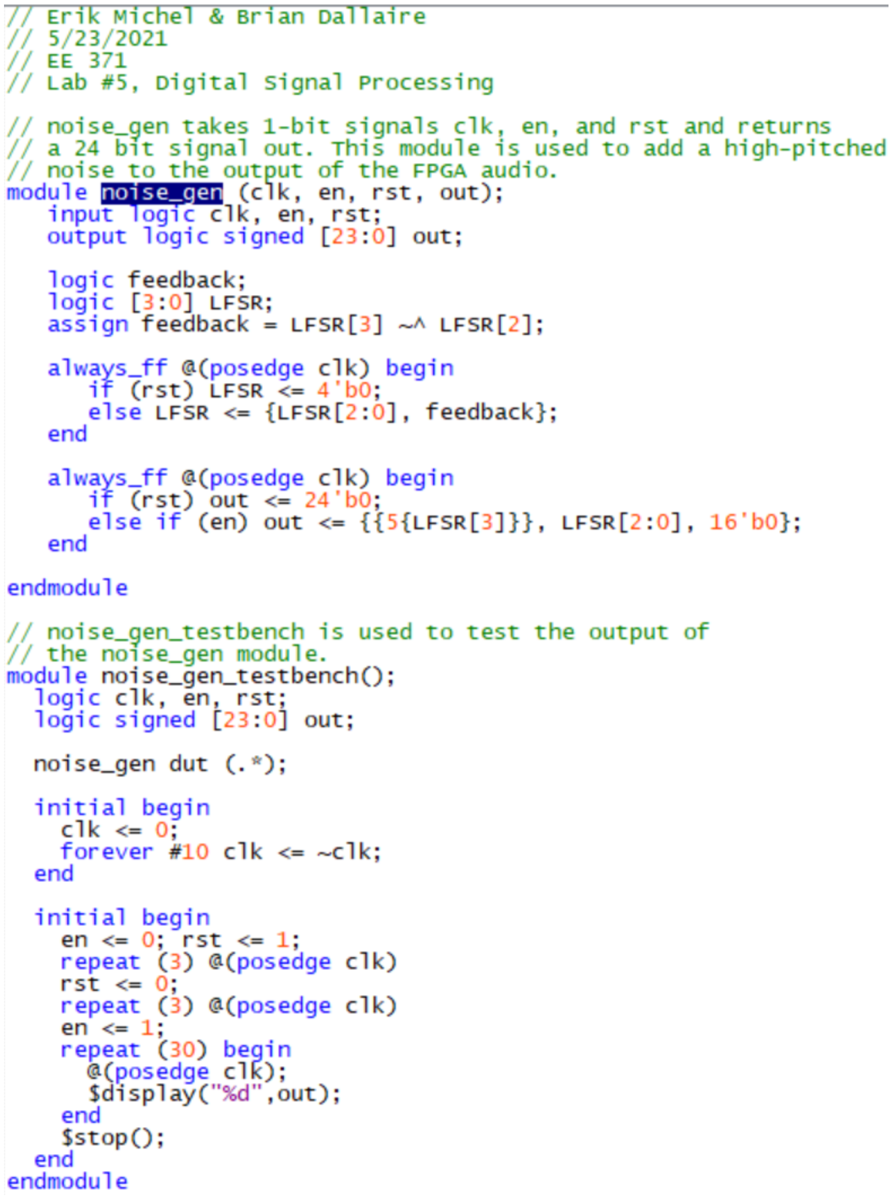
Overall, our lab provided the results we wanted and we believe it is sufficient in covering the requirements of this lab. The special cases were covered for and the primary functions work perfectly.

Appendix

Universal:

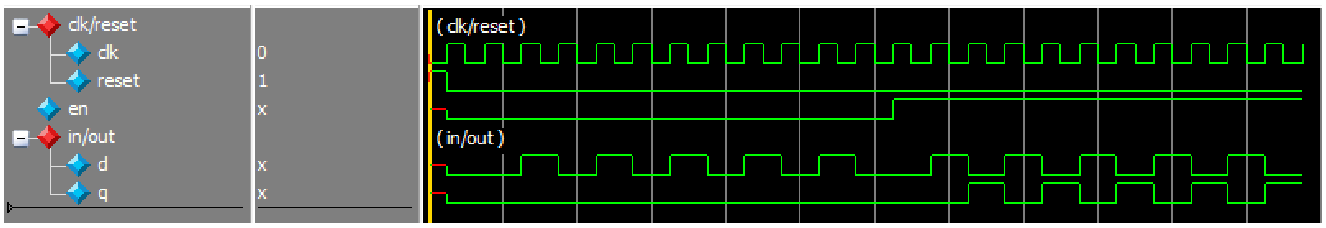
1. noise_gen.sv





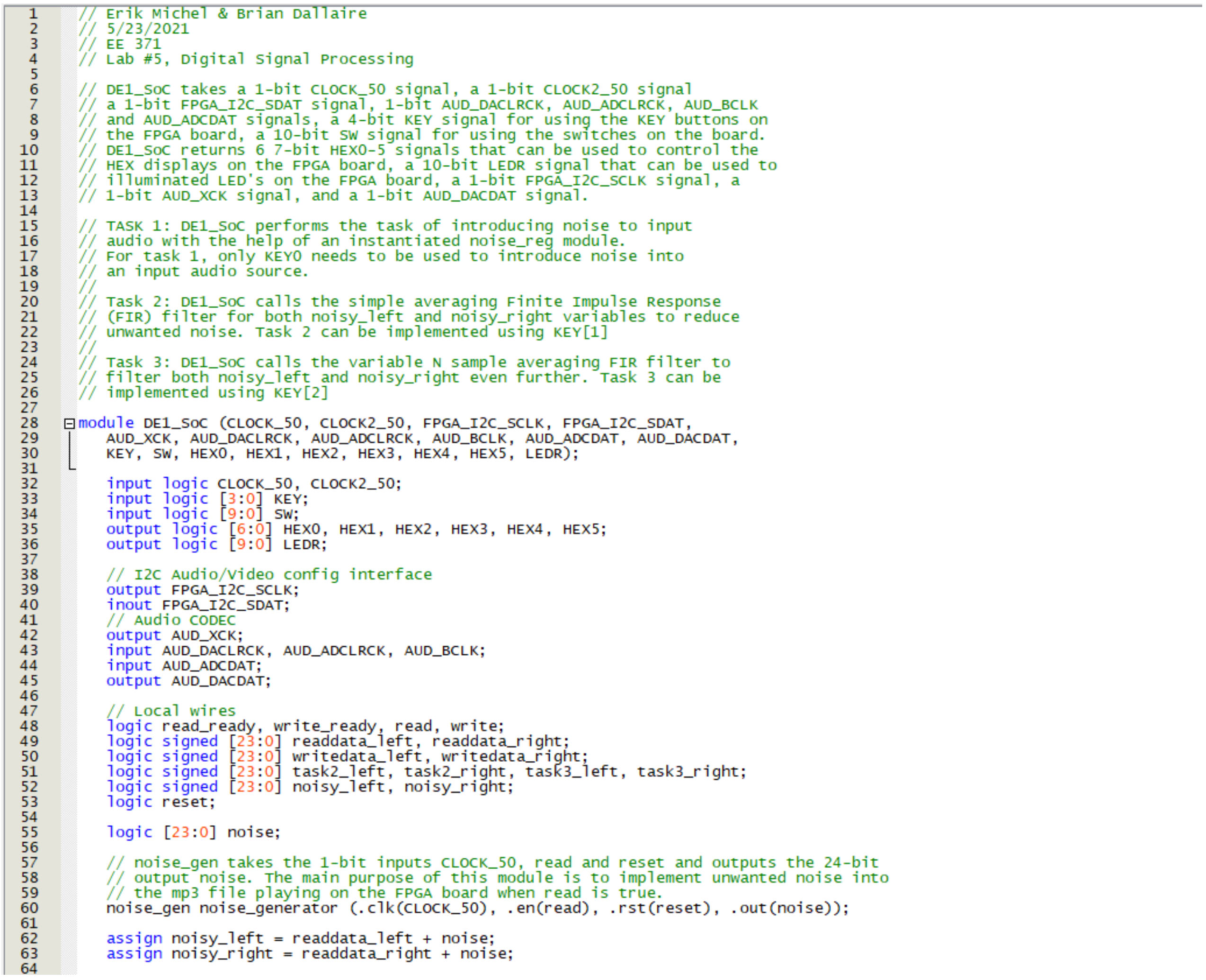
```
// Erik Michel & Brian Dallaire
// 5/23/2021
// EE 371
// Lab #5, Digital Signal Processing

// noise_gen takes 1-bit signals clk, en, and rst and returns
// a 24 bit signal out. This module is used to add a high-pitched
// noise to the output of the FPGA audio.
module noise_gen (clk, en, rst, out);
   input logic clk, en, rst;
   output logic signed [23:0] out;

   logic feedback;
   logic [3:0] LFSR;
   assign feedback = LFSR[3] ~^ LFSR[2];

   always_ff @(posedge clk) begin
      if (rst) LFSR <= 4'b0;
      else LFSR <= {LFSR[2:0], feedback};
   end

   always_ff @(posedge clk) begin
      if (rst) out <= 24'b0;
      else if (en) out <= {{5{LFSR[3]}}, LFSR[2:0], 16'b0};
   end

endmodule

// noise_gen_testbench is used to test the output of
// the noise_gen module.
module noise_gen_testbench();
  logic clk, en, rst;
  logic signed [23:0] out;

  noise_gen dut (.*);

  initial begin
    clk <= 0;
    forever #10 clk <= ~clk;
  end

  initial begin
    en <= 0; rst <= 1;
    repeat (3) @(posedge clk)
    rst <= 0;
    repeat (3) @(posedge clk)
    en <= 1;
    repeat (30) begin
      @(posedge clk);
      $display("%d",out);
    end
    $stop();
  end
endmodule
```

2. wideDFF.sv

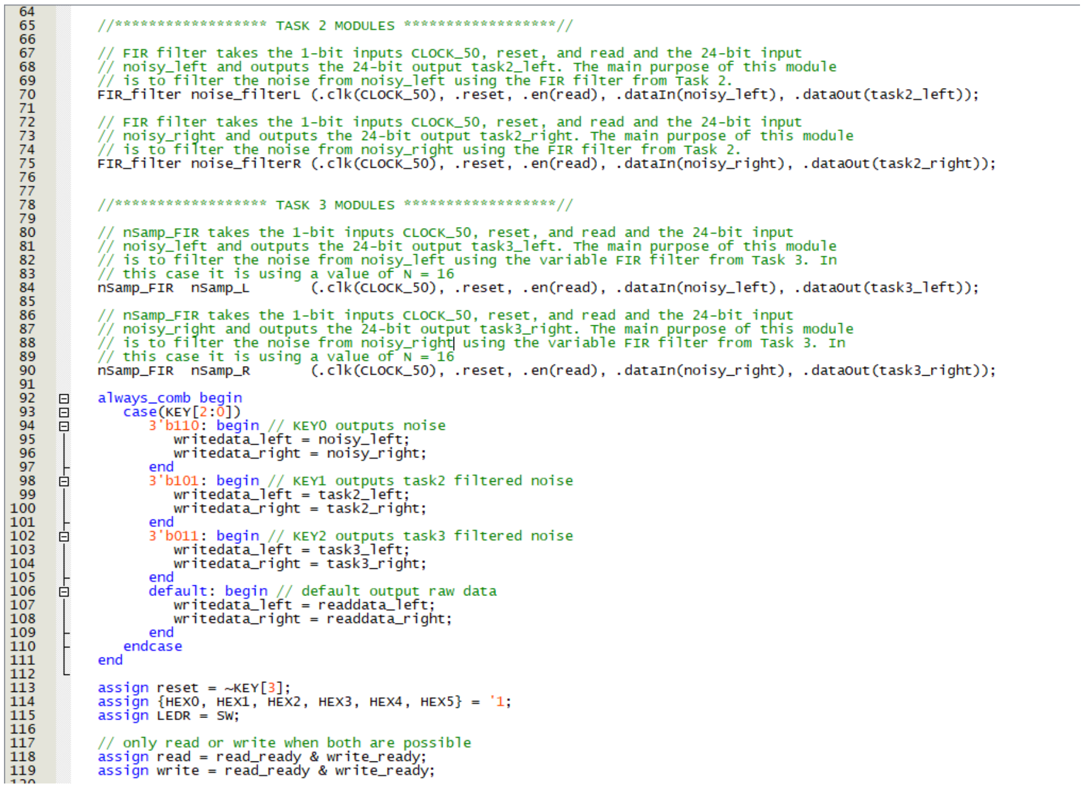


```
// Erik Michel & Brian Dallaire
// 5/23/2021
// EE 371
// Lab #5, Digital Signal Processing

// D_FF takes a 1-bit clk signal, a 1-bit reset signala WIDTH-bit input 'd',
// and returns a WIDTH-bit signal 'q' on the rising edge of the clk signal.
// 'q' is whatever value 'd' was right before the rising edge of the clock.
// when the reset signal is detected, the value of 'q' will become zero
// regardless of the value of 'd'. Also, the value of q will only update
// on the rising edge of the clock if the 'en' signal is ON.
module wideDFF #(parameter WIDTH = 24) (q, d, reset, en, clk);
   output logic [WIDTH-1:0] q;
   input logic  [WIDTH-1:0] d;
   input logic              reset, clk, en;

   always_ff @(posedge clk) begin
      if (reset)
         q <= 0; //on reset, set to 0
      else if (en)
         q <= d; //otherwise out = d
      end
endmodule

// D_FF_testbench tests that given an input d, the noResetFF
// module will output the proper value for q on the rising edge of
// a clk signal
module wideDFF_testbench();
   logic  q;
   logic  d, reset, en, clk;

   wideDFF dut (q, d, reset, en, clk);

   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
   end

   initial begin

      reset <= 1;                              @(posedge clk);
      reset <= 0; d <= 0; en <= 0;             @(posedge clk);
                                               @(posedge clk);

      repeat (5) begin
         d <= 24'b000011110000111100001111;    @(posedge clk);
         d <= 0;                               @(posedge clk);
      end
      en <= 1;                                 @(posedge clk);
            repeat (5) begin
         d <= 24'b000011110000111100001111;    @(posedge clk);
         d <= 0;                               @(posedge clk);
      end

      $stop;

   end
endmodule
```

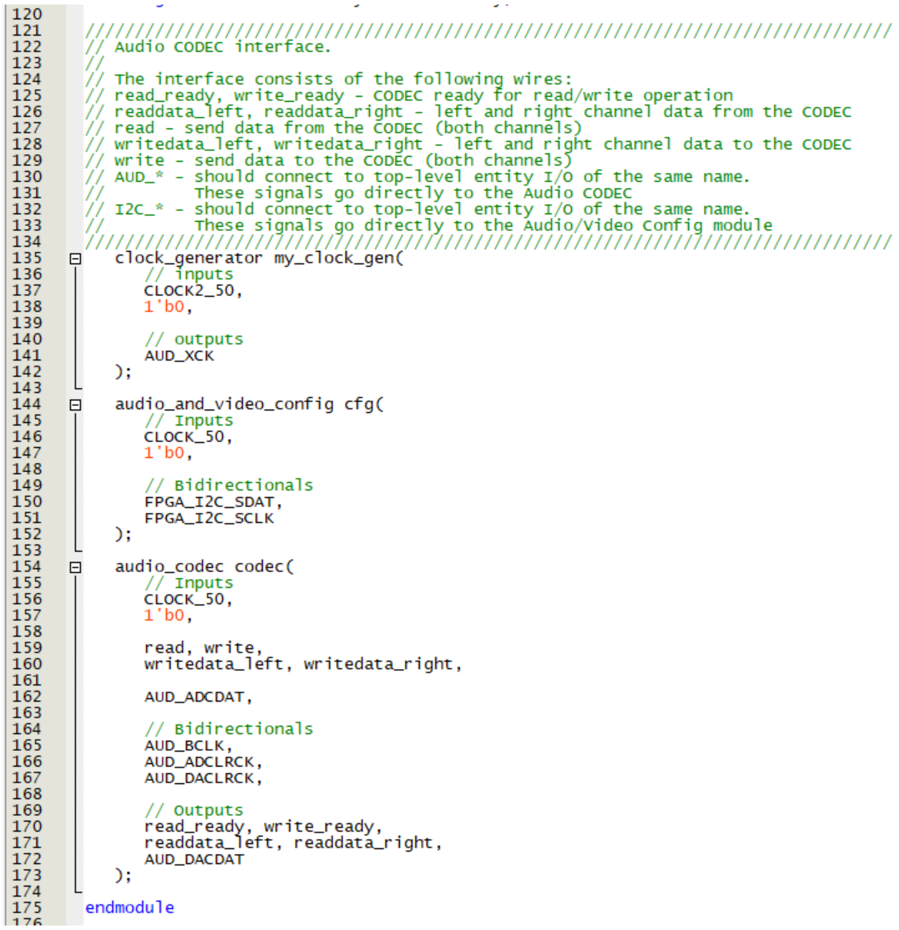
3. DE1_SoC.sv (Task 2 and Task 3 labeled)



```systemverilog
// Erik Michel & Brian Dallaire
// 5/23/2021
// EE 371
// Lab #5, Digital Signal Processing

// DE1_SoC takes a 1-bit CLOCK_50 signal, a 1-bit CLOCK2_50 signal
// a 1-bit FPGA_I2C_SDAT signal, 1-bit AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK
// and AUD_ADCDAT signals, a 4-bit KEY signal for using the KEY buttons on
// the FPGA board, a 10-bit SW signal for using the switches on the board.
// DE1_SoC returns 6 7-bit HEX0-5 signals that can be used to control the
// HEX displays on the FPGA board, a 10-bit LEDR signal that can be used to
// illuminated LED's on the FPGA board, a 1-bit FPGA_I2C_SCLK signal, a
// 1-bit AUD_XCK signal, and a 1-bit AUD_DACDAT signal.

// TASK 1: DE1_SoC performs the task of introducing noise to input
// audio with the help of an instantiated noise_reg module.
// For task 1, only KEY0 needs to be used to introduce noise into
// an input audio source.
//
// Task 2: DE1_SoC calls the simple averaging Finite Impulse Response
// (FIR) filter for both noisy_left and noisy_right variables to reduce
// unwanted noise. Task 2 can be implemented using KEY[1]
//
// Task 3: DE1_SoC calls the variable N sample averaging FIR filter to
// filter both noisy_left and noisy_right even further. Task 3 can be
// implemented using KEY[2]

module DE1_SoC (CLOCK_50, CLOCK2_50, FPGA_I2C_SCLK, FPGA_I2C_SDAT,
    AUD_XCK, AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK, AUD_ADCDAT, AUD_DACDAT,
    KEY, SW, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, LEDR);

    input logic CLOCK_50, CLOCK2_50;
    input logic [3:0] KEY;
    input logic [9:0] SW;
    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    output logic [9:0] LEDR;

    // I2C Audio/Video config interface
    output FPGA_I2C_SCLK;
    inout FPGA_I2C_SDAT;
    // Audio CODEC
    output AUD_XCK;
    input AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK;
    input AUD_ADCDAT;
    output AUD_DACDAT;

    // Local wires
    logic read_ready, write_ready, read, write;
    logic signed [23:0] readdata_left, readdata_right;
    logic signed [23:0] writedata_left, writedata_right;
    logic signed [23:0] task2_left, task2_right, task3_left, task3_right;
    logic signed [23:0] noisy_left, noisy_right;
    logic reset;

    logic [23:0] noise;

    // noise_gen takes the 1-bit inputs CLOCK_50, read and reset and outputs the 24-bit
    // output noise. The main purpose of this module is to implement unwanted noise into
    // the mp3 file playing on the FPGA board when read is true.
    noise_gen noise_generator (.clk(CLOCK_50), .en(read), .rst(reset), .out(noise));

    assign noisy_left = readdata_left + noise;
    assign noisy_right = readdata_right + noise;

```



```verilog

//****************** TASK 2 MODULES ******************//

// FIR filter takes the 1-bit inputs CLOCK_50, reset, and read and the 24-bit input
// noisy_left and outputs the 24-bit output task2_left. The main purpose of this module
// is to filter the noise from noisy_left using the FIR filter from Task 2.
FIR_filter noise_filterL (.clk(CLOCK_50), .reset, .en(read), .dataIn(noisy_left), .dataOut(task2_left));

// FIR filter takes the 1-bit inputs CLOCK_50, reset, and read and the 24-bit input
// noisy_right and outputs the 24-bit output task2_right. The main purpose of this module
// is to filter the noise from noisy_right using the FIR filter from Task 2.
FIR_filter noise_filterR (.clk(CLOCK_50), .reset, .en(read), .dataIn(noisy_right), .dataOut(task2_right));


//****************** TASK 3 MODULES ******************//

// nSamp_FIR takes the 1-bit inputs CLOCK_50, reset, and read and the 24-bit input
// noisy_left and outputs the 24-bit output task3_left. The main purpose of this module
// is to filter the noise from noisy_left using the variable FIR filter from Task 3. In
// this case it is using a value of N = 16
nSamp_FIR  nSamp_L       (.clk(CLOCK_50), .reset, .en(read), .dataIn(noisy_left), .dataOut(task3_left));

// nSamp_FIR takes the 1-bit inputs CLOCK_50, reset, and read and the 24-bit input
// noisy_right and outputs the 24-bit output task3_right. The main purpose of this module
// is to filter the noise from noisy_right using the variable FIR filter from Task 3. In
// this case it is using a value of N = 16
nSamp_FIR  nSamp_R       (.clk(CLOCK_50), .reset, .en(read), .dataIn(noisy_right), .dataOut(task3_right));

always_comb begin
   case(KEY[2:0])
      3'b110: begin // KEY0 outputs noise
         writedata_left = noisy_left;
         writedata_right = noisy_right;
      end
      3'b101: begin // KEY1 outputs task2 filtered noise
         writedata_left = task2_left;
         writedata_right = task2_right;
      end
      3'b011: begin // KEY2 outputs task3 filtered noise
         writedata_left = task3_left;
         writedata_right = task3_right;
      end
      default: begin // default output raw data
         writedata_left = readdata_left;
         writedata_right = readdata_right;
      end
   endcase
end

assign reset = ~KEY[3];
assign {HEX0, HEX1, HEX2, HEX3, HEX4, HEX5} = '1;
assign LEDR = SW;

// only read or write when both are possible
assign read = read_ready & write_ready;
assign write = read_ready & write_ready;
```

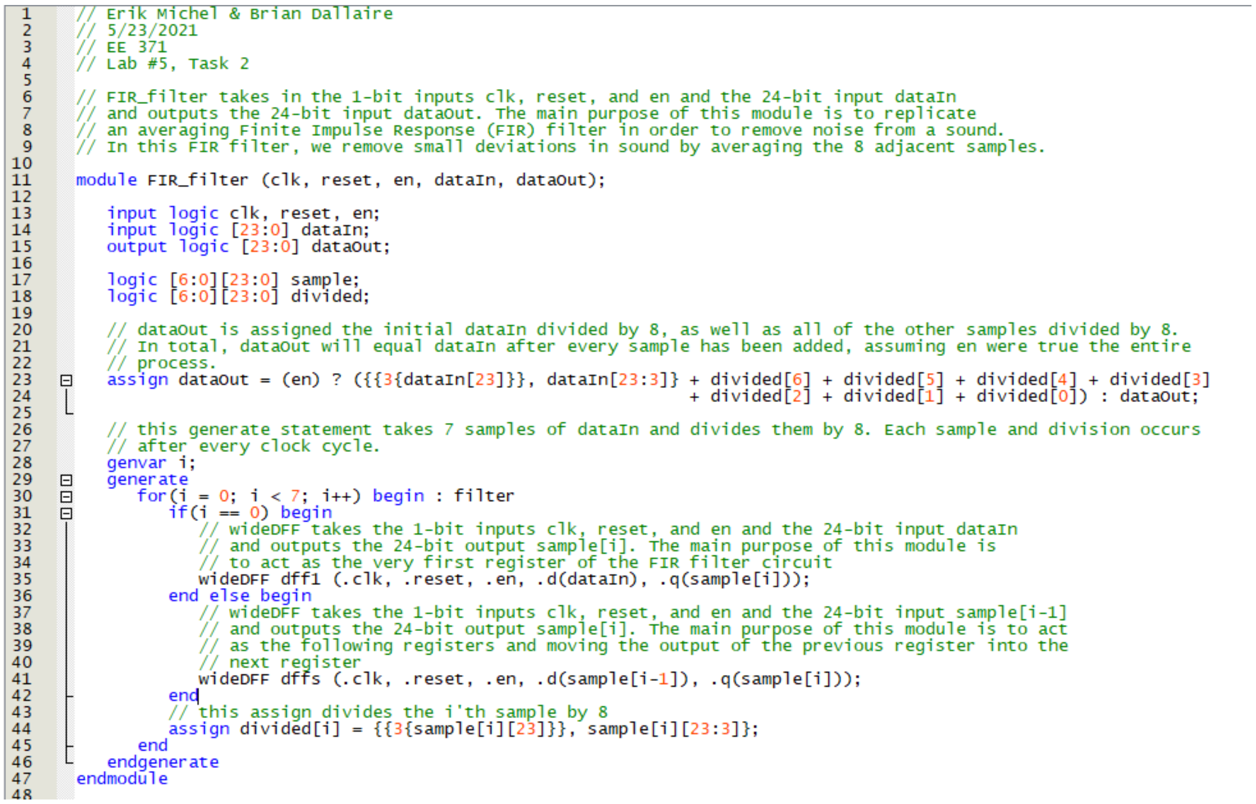


```verilog

/////////////////////////////////////////////////////////////////////////////////
// Audio CODEC interface.
//
// The interface consists of the following wires:
// read_ready, write_ready - CODEC ready for read/write operation
// readdata_left, readdata_right - left and right channel data from the CODEC
// read - send data from the CODEC (both channels)
// writedata_left, writedata_right - left and right channel data to the CODEC
// write - send data to the CODEC (both channels)
// AUD_* - should connect to top-level entity I/O of the same name.
//         These signals go directly to the Audio CODEC
// I2C_* - should connect to top-level entity I/O of the same name.
//         These signals go directly to the Audio/Video Config module
/////////////////////////////////////////////////////////////////////////////////
   clock_generator my_clock_gen(
      // inputs
      CLOCK2_50,
      1'b0,

      // outputs
      AUD_XCK
   );

   audio_and_video_config cfg(
      // Inputs
      CLOCK_50,
      1'b0,

      // Bidirectionals
      FPGA_I2C_SDAT,
      FPGA_I2C_SCLK
   );

   audio_codec codec(
      // Inputs
      CLOCK_50,
      1'b0,

      read, write,
      writedata_left, writedata_right,

      AUD_ADCDAT,

      // Bidirectionals
      AUD_BCLK,
      AUD_ADCLRCK,
      AUD_DACLRCK,

      // Outputs
      read_ready, write_ready,
      readdata_left, readdata_right,
      AUD_DACDAT
   );

endmodule
```

Task 2:

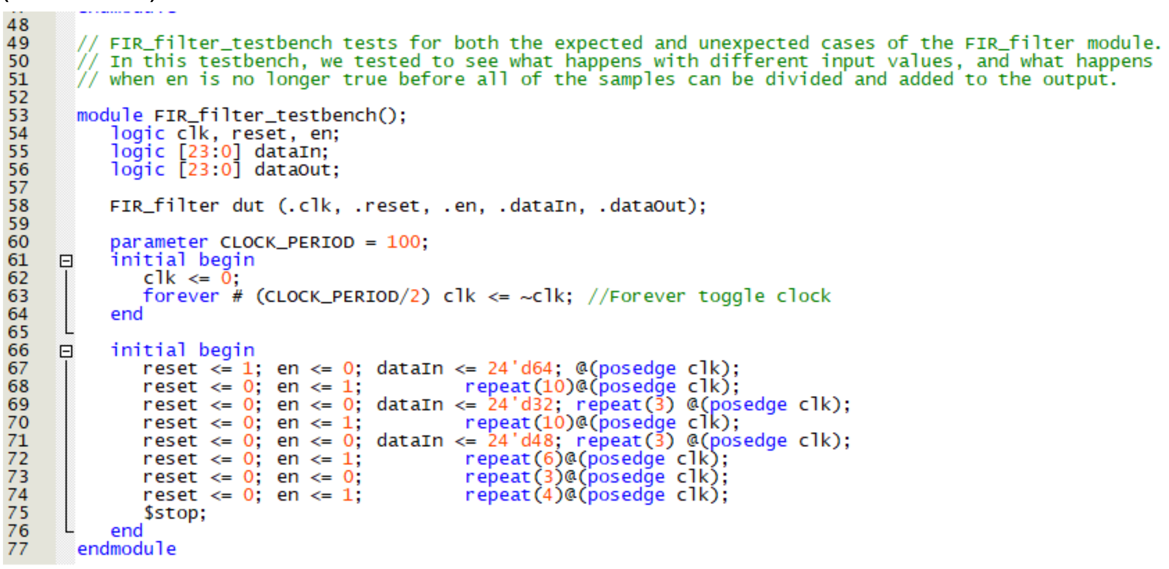
4. FIR_filter.sv

(code)



```
// Erik Michel & Brian Dallaire
// 5/23/2021
// EE 371
// Lab #5, Task 2

// FIR_filter takes in the 1-bit inputs clk, reset, and en and the 24-bit input dataIn
// and outputs the 24-bit input dataOut. The main purpose of this module is to replicate
// an averaging Finite Impulse Response (FIR) filter in order to remove noise from a sound.
// In this FIR filter, we remove small deviations in sound by averaging the 8 adjacent samples.

module FIR_filter (clk, reset, en, dataIn, dataOut);

   input logic clk, reset, en;
   input logic [23:0] dataIn;
   output logic [23:0] dataOut;

   logic [6:0][23:0] sample;
   logic [6:0][23:0] divided;

   // dataOut is assigned the initial dataIn divided by 8, as well as all of the other samples divided by 8.
   // In total, dataOut will equal dataIn after every sample has been added, assuming en were true the entire
   // process.
   assign dataOut = (en) ? ({{3{dataIn[23]}}, dataIn[23:3]} + divided[6] + divided[5] + divided[4] + divided[3]
                                                           + divided[2] + divided[1] + divided[0]) : dataOut;

   // this generate statement takes 7 samples of dataIn and divides them by 8. Each sample and division occurs
   // after every clock cycle.
   genvar i;
   generate
      for(i = 0; i < 7; i++) begin : filter
         if(i == 0) begin
            // wideDFF takes the 1-bit inputs clk, reset, and en and the 24-bit input dataIn
            // and outputs the 24-bit output sample[i]. The main purpose of this module is
            // to act as the very first register of the FIR filter circuit
            wideDFF dff1 (.clk, .reset, .en, .d(dataIn), .q(sample[i]));
         end else begin
            // wideDFF takes the 1-bit inputs clk, reset, and en and the 24-bit input sample[i-1]
            // and outputs the 24-bit output sample[i]. The main purpose of this module is to act
            // as the following registers and moving the output of the previous register into the
            // next register
            wideDFF dffs (.clk, .reset, .en, .d(sample[i-1]), .q(sample[i]));
         end
         // this assign divides the i'th sample by 8
         assign divided[i] = {{3{sample[i][23]}}, sample[i][23:3]};
      end
   endgenerate
endmodule
```

(testbench)

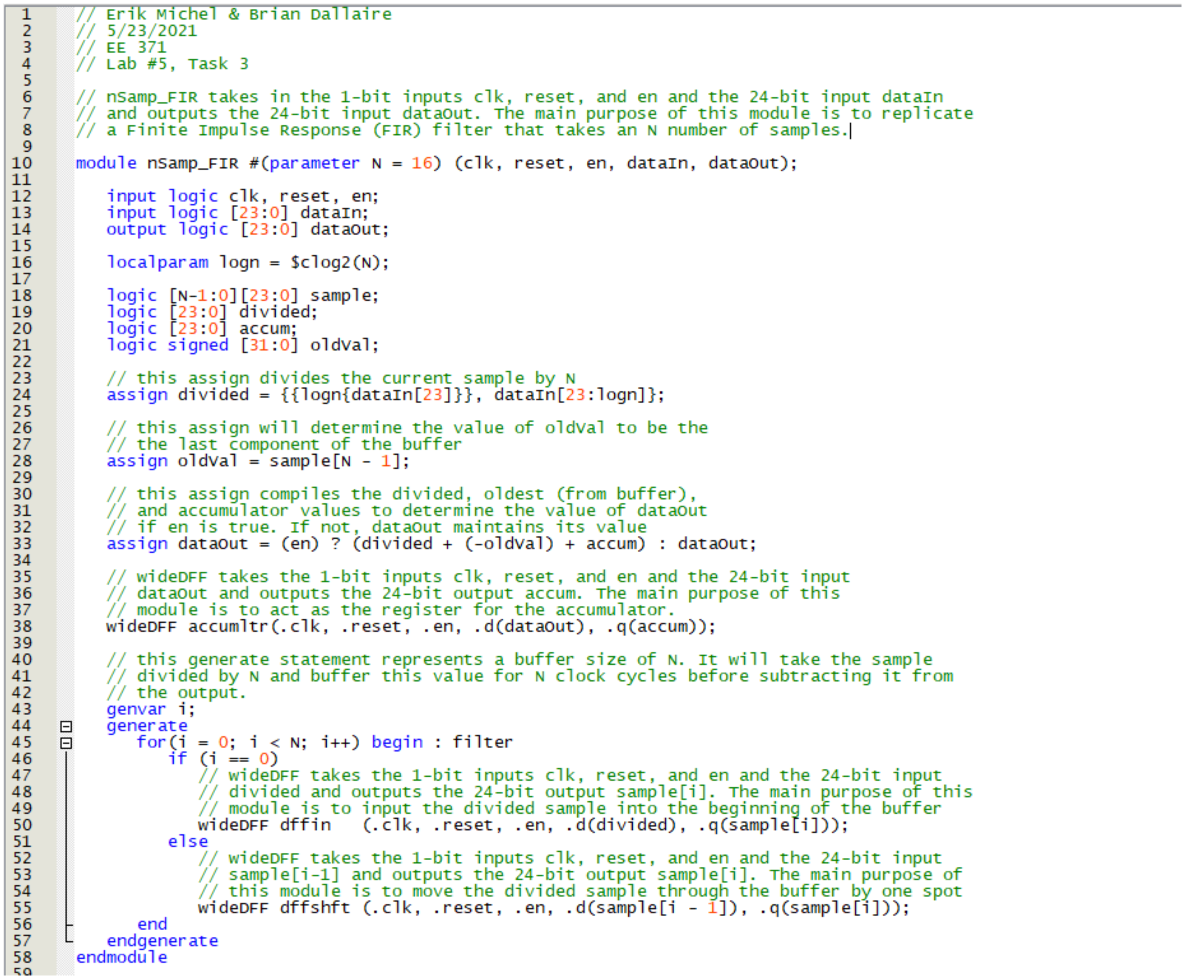


```

// FIR_filter_testbench tests for both the expected and unexpected cases of the FIR_filter module.
// In this testbench, we tested to see what happens with different input values, and what happens
// when en is no longer true before all of the samples can be divided and added to the output.

module FIR_filter_testbench();
   logic clk, reset, en;
   logic [23:0] dataIn;
   logic [23:0] dataOut;

   FIR_filter dut (.clk, .reset, .en, .dataIn, .dataOut);

   parameter CLOCK_PERIOD = 100;
   initial begin
      clk <= 0;
      forever # (CLOCK_PERIOD/2) clk <= ~clk; //Forever toggle clock
   end

   initial begin
      reset <= 1; en <= 0; dataIn <= 24'd64; @(posedge clk);
      reset <= 0; en <= 1;         repeat(10)@(posedge clk);
      reset <= 0; en <= 0; dataIn <= 24'd32; repeat(3) @(posedge clk);
      reset <= 0; en <= 1;         repeat(10)@(posedge clk);
      reset <= 0; en <= 0; dataIn <= 24'd48; repeat(3) @(posedge clk);
      reset <= 0; en <= 1;         repeat(6)@(posedge clk);
      reset <= 0; en <= 0;         repeat(3)@(posedge clk);
      reset <= 0; en <= 1;         repeat(4)@(posedge clk);
      $stop;
   end
endmodule
```

Task 3:

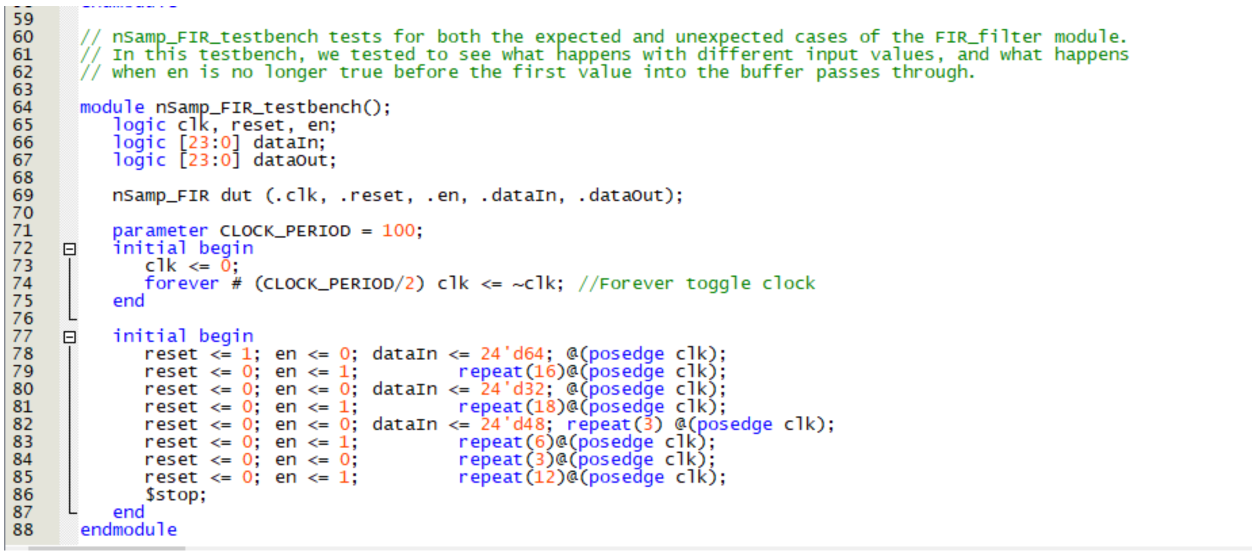
5. nSamp_FIR.sv

(code)



```
// Erik Michel & Brian Dallaire
// 5/23/2021
// EE 371
// Lab #5, Task 3

// nSamp_FIR takes in the 1-bit inputs clk, reset, and en and the 24-bit input dataIn
// and outputs the 24-bit input dataOut. The main purpose of this module is to replicate
// a Finite Impulse Response (FIR) filter that takes an N number of samples.

module nSamp_FIR #(parameter N = 16) (clk, reset, en, dataIn, dataOut);

   input logic clk, reset, en;
   input logic [23:0] dataIn;
   output logic [23:0] dataOut;

   localparam logn = $clog2(N);

   logic [N-1:0][23:0] sample;
   logic [23:0] divided;
   logic [23:0] accum;
   logic signed [31:0] oldVal;

   // this assign divides the current sample by N
   assign divided = {{logn{dataIn[23]}}, dataIn[23:logn]};

   // this assign will determine the value of oldVal to be the
   // the last component of the buffer
   assign oldVal = sample[N - 1];

   // this assign compiles the divided, oldest (from buffer),
   // and accumulator values to determine the value of dataOut
   // if en is true. If not, dataOut maintains its value
   assign dataOut = (en) ? (divided + (-oldVal) + accum) : dataOut;

   // wideDFF takes the 1-bit inputs clk, reset, and en and the 24-bit input
   // dataOut and outputs the 24-bit output accum. The main purpose of this
   // module is to act as the register for the accumulator.
   wideDFF accumltr(.clk, .reset, .en, .d(dataOut), .q(accum));

   // this generate statement represents a buffer size of N. It will take the sample
   // divided by N and buffer this value for N clock cycles before subtracting it from
   // the output.
   genvar i;
   generate
      for(i = 0; i < N; i++) begin : filter
         if (i == 0)
            // wideDFF takes the 1-bit inputs clk, reset, and en and the 24-bit input
            // divided and outputs the 24-bit output sample[i]. The main purpose of this
            // module is to input the divided sample into the beginning of the buffer
            wideDFF dffin   (.clk, .reset, .en, .d(divided), .q(sample[i]));
         else
            // wideDFF takes the 1-bit inputs clk, reset, and en and the 24-bit input
            // sample[i-1] and outputs the 24-bit output sample[i]. The main purpose of
            // this module is to move the divided sample through the buffer by one spot
            wideDFF dffshft (.clk, .reset, .en, .d(sample[i - 1]), .q(sample[i]));
      end
   endgenerate
endmodule
```

(testbench)



```
// nSamp_FIR_testbench tests for both the expected and unexpected cases of the FIR_filter module.
// In this testbench, we tested to see what happens with different input values, and what happens
// when en is no longer true before the first value into the buffer passes through.

module nSamp_FIR_testbench();
   logic clk, reset, en;
   logic [23:0] dataIn;
   logic [23:0] dataOut;

   nSamp_FIR dut (.clk, .reset, .en, .dataIn, .dataOut);

   parameter CLOCK_PERIOD = 100;
   initial begin
      clk <= 0;
      forever # (CLOCK_PERIOD/2) clk <= ~clk; //Forever toggle clock
   end

   initial begin
      reset <= 1; en <= 0; dataIn <= 24'd64; @(posedge clk);
      reset <= 0; en <= 1;          repeat(16)@(posedge clk);
      reset <= 0; en <= 0; dataIn <= 24'd32; @(posedge clk);
      reset <= 0; en <= 1;          repeat(18)@(posedge clk);
      reset <= 0; en <= 0; dataIn <= 24'd48; repeat(3) @(posedge clk);
      reset <= 0; en <= 1;          repeat(6)@(posedge clk);
      reset <= 0; en <= 0;          repeat(3)@(posedge clk);
      reset <= 0; en <= 1;          repeat(12)@(posedge clk);
      $stop;
   end
endmodule
```