Cynthia Li, 1839952
EE 371
May 23, 2021

Lab 5 Report

Procedure

This lab is about understanding the sampling, coding and decoding of audio files using the audio coder/decoder function on the FPGA board. We use the data from the CODEC for signal processing including introducing noise and filtering out noise. There were particularly two different filters we chose to implement and test to see their effectiveness. In all, there were 3 tasks to do.

Task 1: This task is about testing the audio interface. We just follow the lab manual and test the provided starter code. From the spec, we know that CODEC will only read samples from its buffer when it is ready to both read and write. This is very important since we want to record whatever the microphone's input, so read_data needs to be passed to write_data. Below is the top module block diagram for task 1.

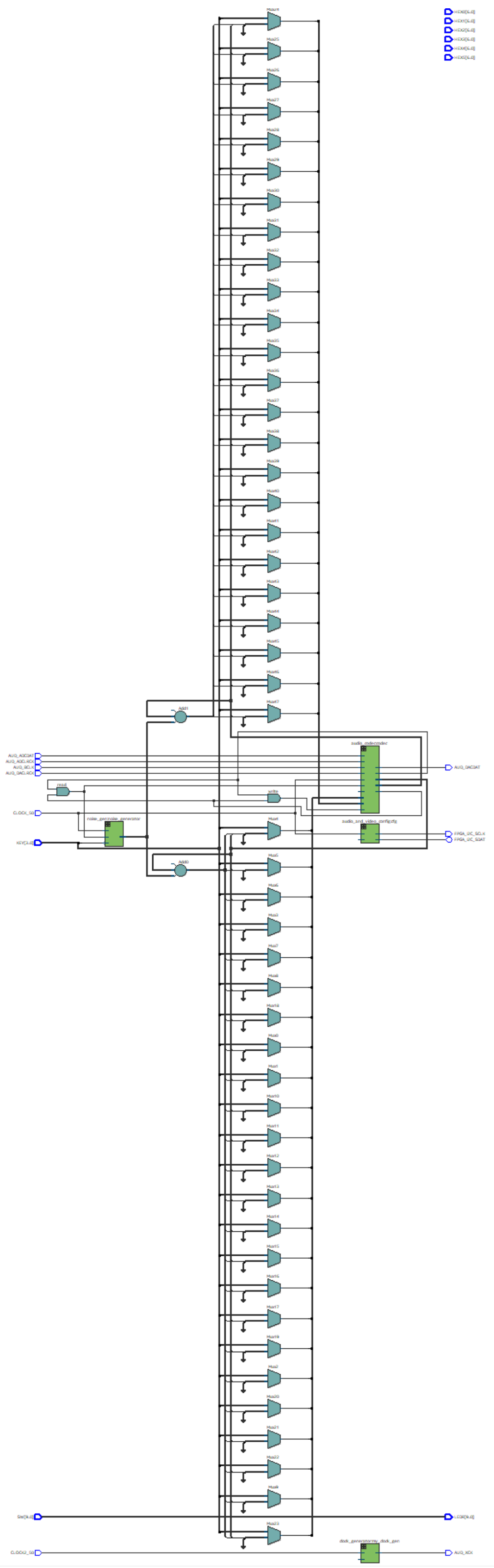
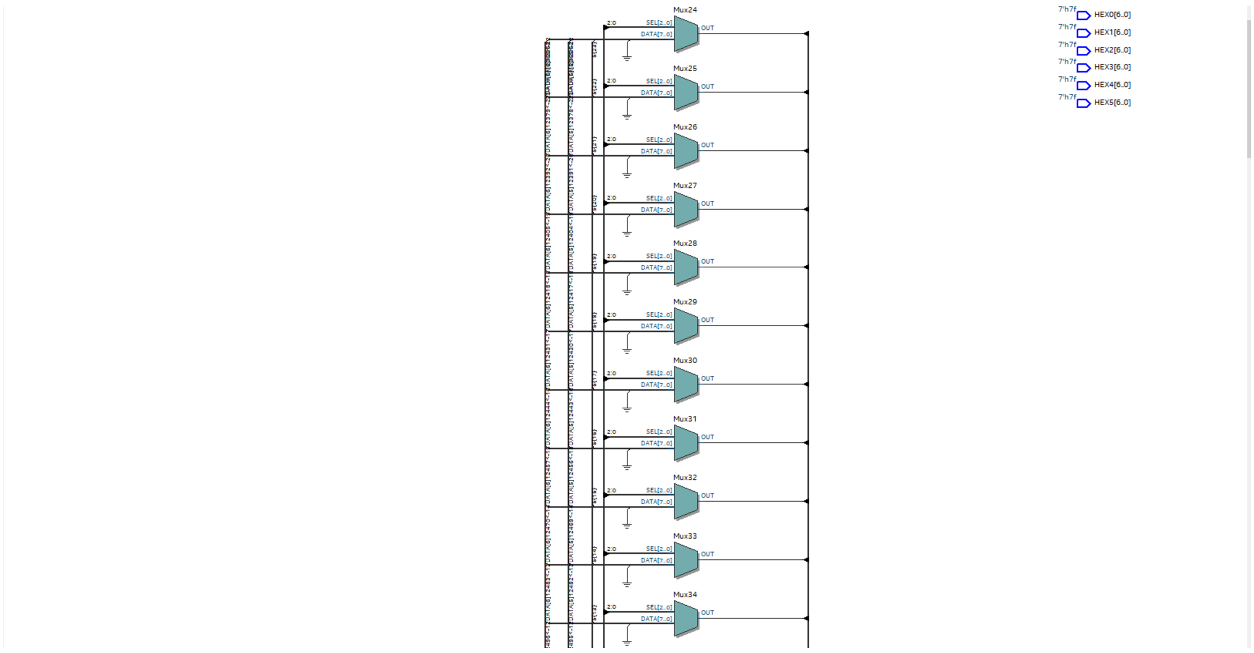
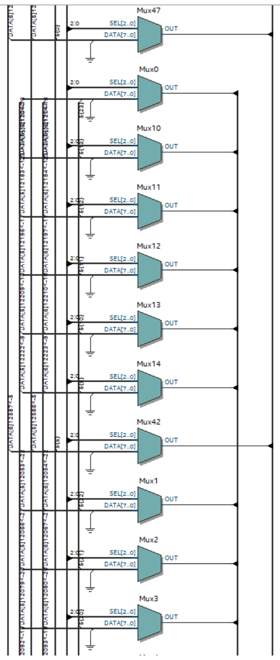
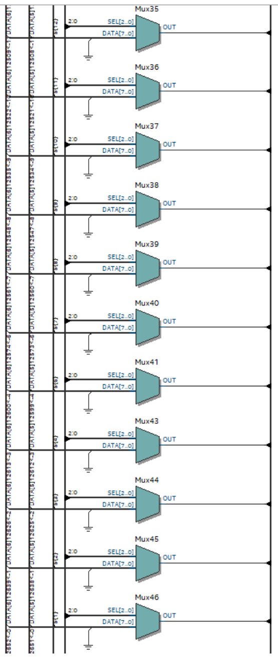


Fig. 1 Task 1 top module block diagram

Task 2: This task is different from task 1 in which we just need to record the audio with noise, instead we will use a filter to remove some noise. According to the spec, the filter should work by averaging the adjacent 8 samples. We develop a module called sample8_filter that takes in a 24-bit input and outputs a 24-bit value equal to the average of the 8 values near this input. Below is the top module block diagram for task 2.




Mux47
Mux0
Mux10
Mux11
Mux12
Mux13
Mux14
Mux42
Mux1
Mux2
Mux3
SEL[2..0]
DATA[7..0]
OUT


Mux35
Mux36
Mux37
Mux38
Mux39
Mux40
Mux41
Mux43
Mux44
Mux45
Mux46
SEL[2..0]
DATA[7..0]
OUT

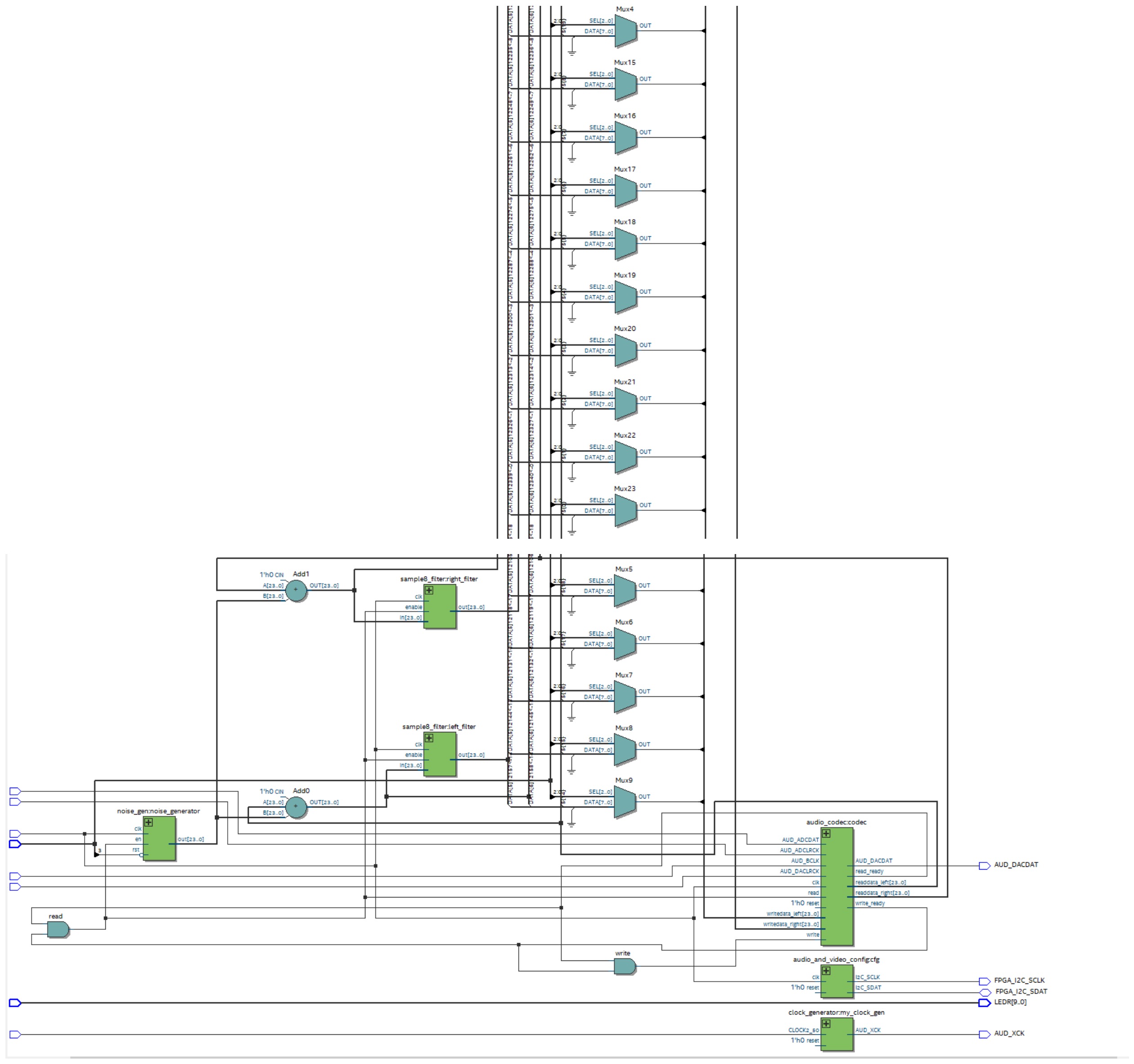


Fig. 2 Task 2 top block diagram

Task 3: In this task, we want to implement a more general filter that allows us to pick any N. The step to solve this task is first to implement a shift-register-like fifo buffer, use combinational logic to connect the relationship between each variable as given by the lab specification, use a DFF to delay the output one cycle late for the accumulator, and lastly wire everything up. Below is the top block diagram for this task.

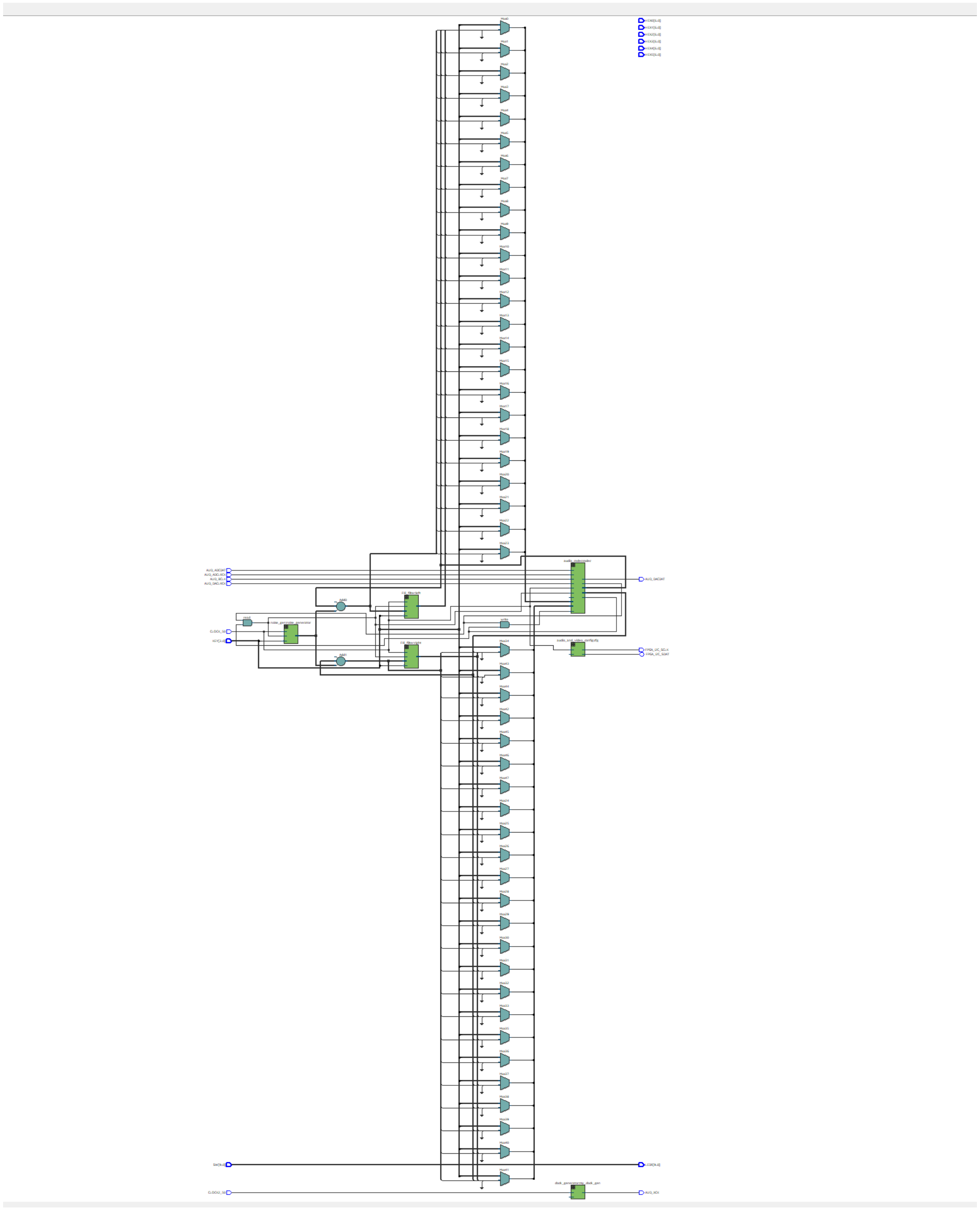


Fig. 3 Task 3 top block diagram

Result:

Task 1:

There is no code written for this part. We simply simulated the given code noise_gen to understand how the noise is generated and how it will affect the music. It produced the waveform as below in Fig. 4 and Fig 5

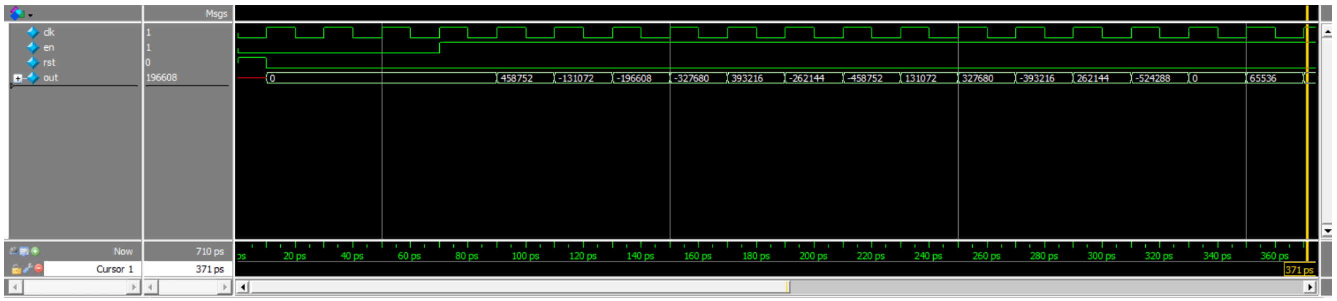


Fig. 4 noise_gen simulation

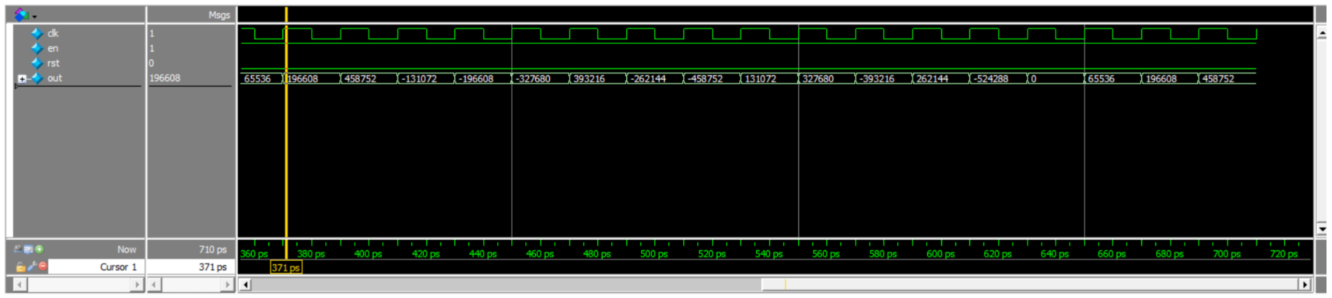
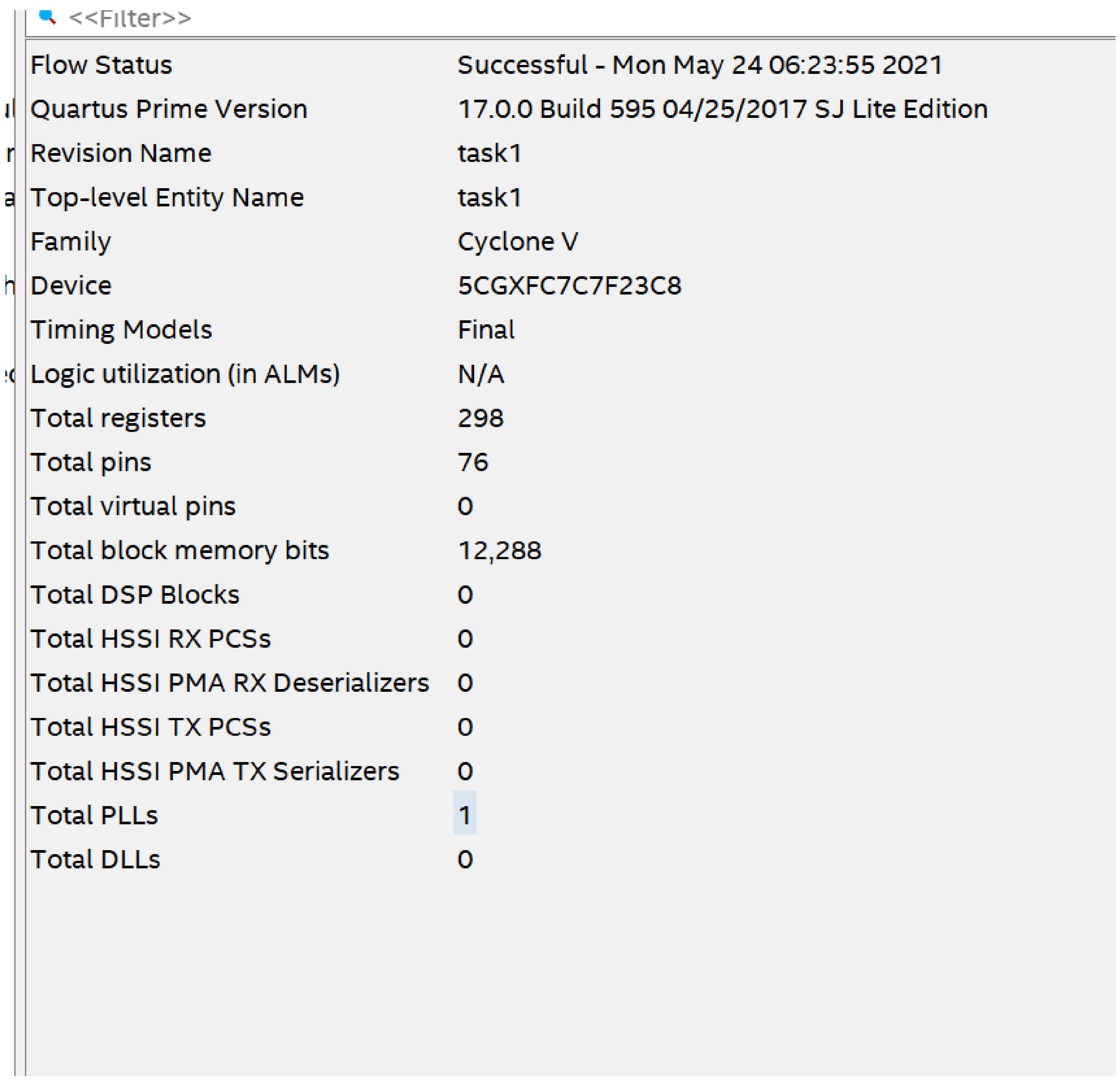


Fig. 5 noise_gen simulation

We simply want to see whether rst turns off noise, and let it start again, whether en starts generates noises. We expect to see some high magnitude value being generated as noise. From the simulation, we can see that the output was as expected so desired noise can be successfully produced.

Below is the flow summary of task1.


<<Filter>>

Flow Status	Successful - Mon May 24 06:23:55 2021
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	task1
Top-level Entity Name	task1
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	298
Total pins	76
Total virtual pins	0
Total block memory bits	12,288
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1
Total DLLs	0

Fig. 6 Task 1 flow summary

Task 2:
For task 2, we want to implement a filter to average the value of the nearest 8 points to smooth the music. We used an array to hold the values first, then sum up the divided newest input and the values in the array, and finally output the result as a 24 bit value.

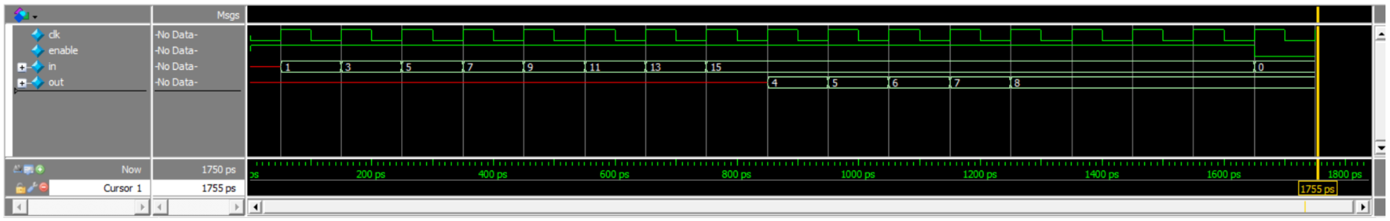
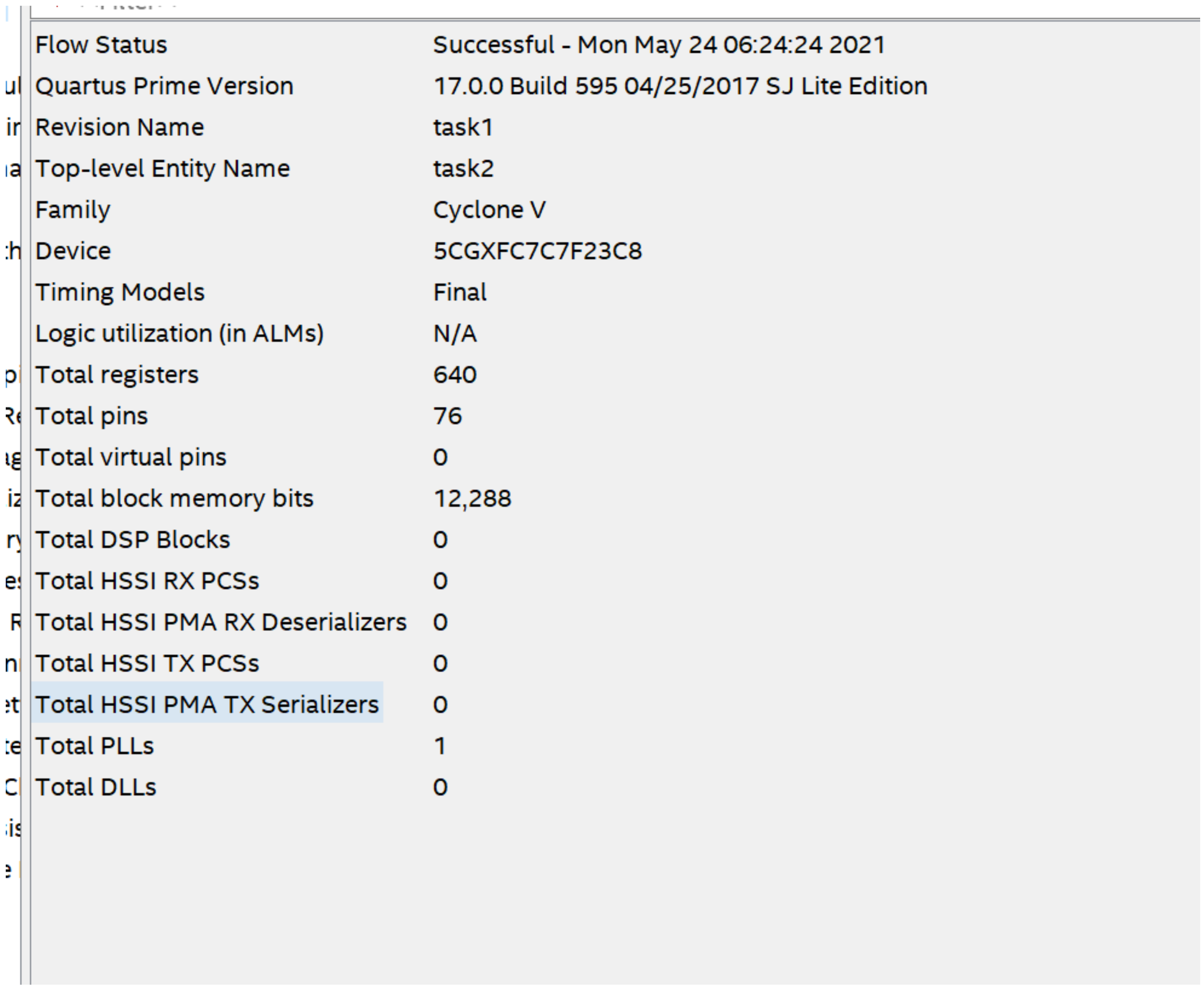


Fig. 7 sample8_filter_testbench simulation

We want to test if the filter can output the averaged value of the nearest 8 indexes when enable = 1. From the simulation, we observe that after 8 cycles (when the 8 samples have been collected) it gradually averages the samples and stabilizes at the correct answer 8. We also observe that out becomes 0 when enable is turned off, which is what we've expected. Therefore, we think this filter has correct functionality.

Lastly, we use the filter to process the noisy signal in the top module. We wire noisy_left as input, task2_left as output for the left signal, and noisy_right as input, task2_right as output for the right signal. We expect the filter to reduce the noise a bit, as shown in the demo.

Below is the flow summary of task2.



Flow Status	Successful - Mon May 24 06:24:24 2021
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	task1
Top-level Entity Name	task2
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	640
Total pins	76
Total virtual pins	0
Total block memory bits	12,288
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1
Total DLLs	0

Fig. 8 Task 2 flow summary

Task 3:

For this task, we want to pass the input to the fifo buffer after dividing it by N and then accumulate them to the output of the filter.

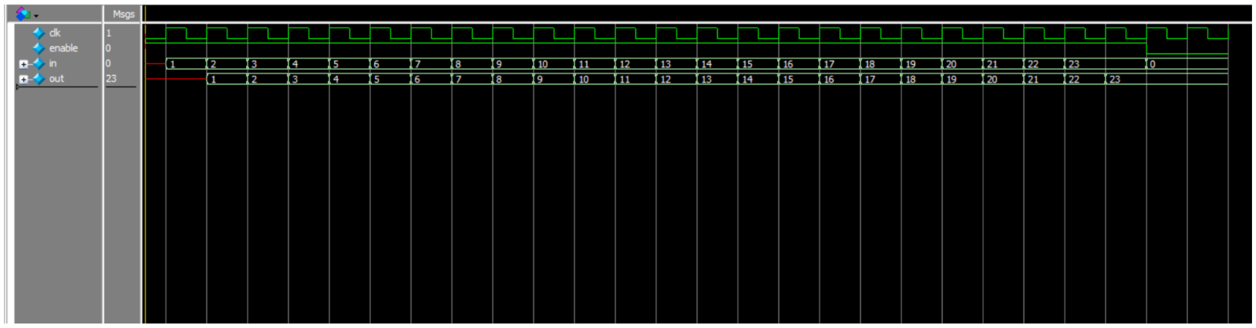

Msgs
clk 1
enable 0
in 0
out 23
in: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 0
out: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

Fig. 9 delay_testbench simulation

Delay.sv is just a DFF, so the expected output waveform should be that output is one clock cycle delayed than input. The above waveform matches our expected output waveform.

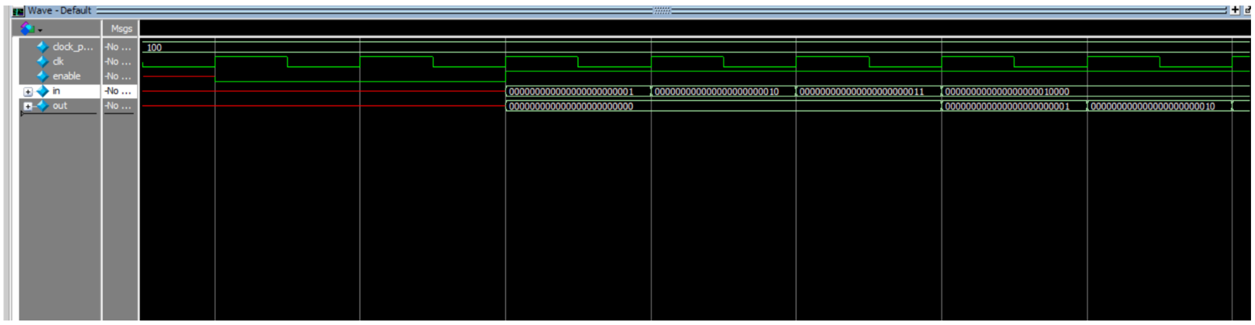


Fig. 10 FIR_filter_testbench simulation

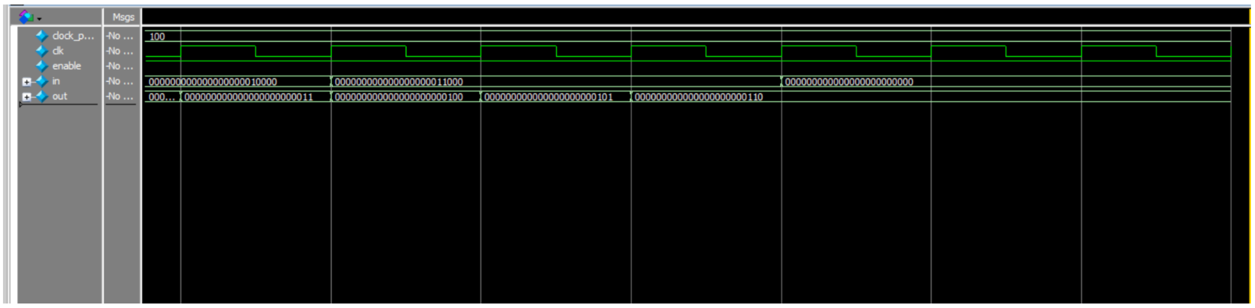
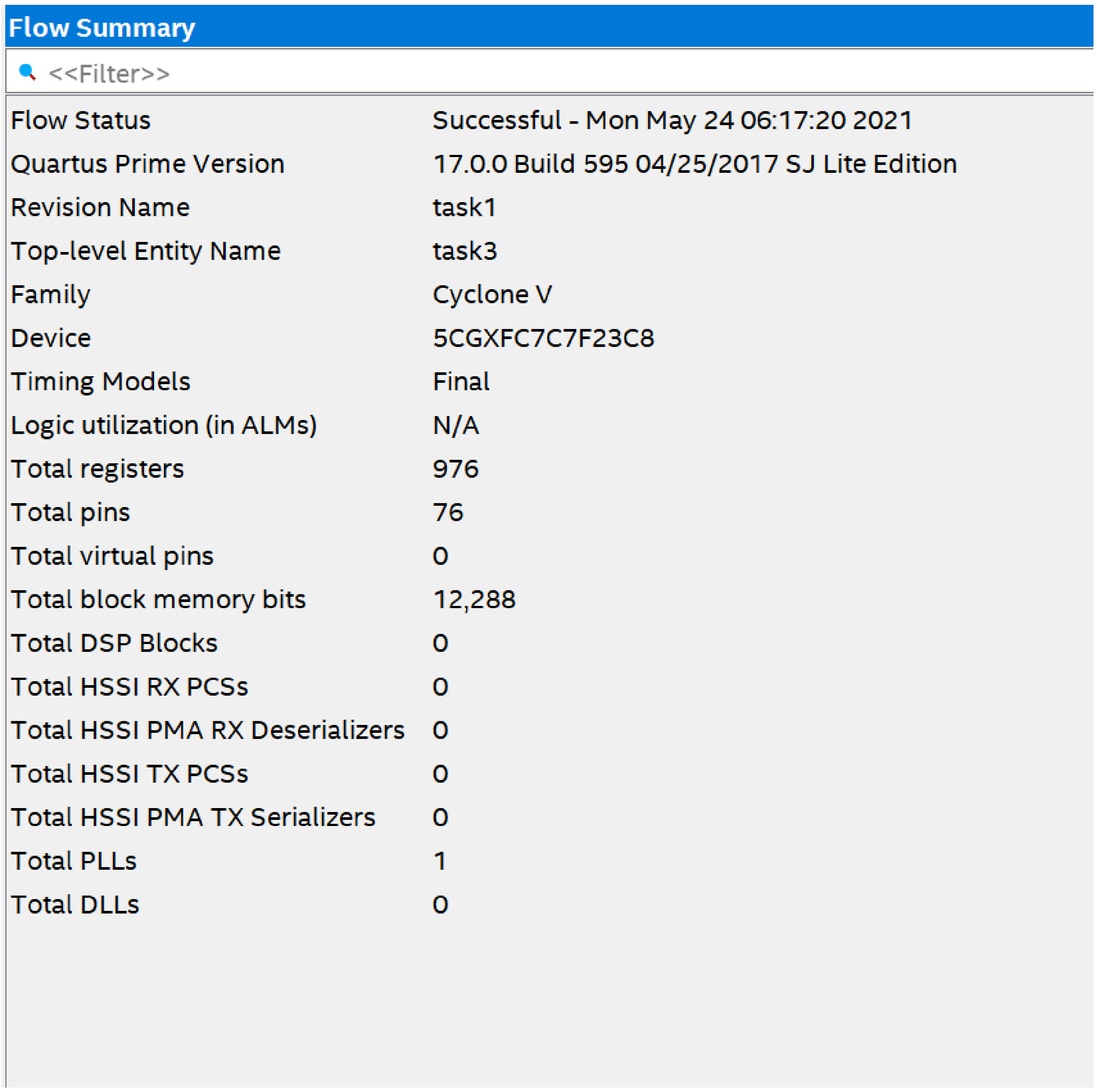


Fig. 11 FIR_filter_testbench simulation

For FIR_filter_testbench, I am testing the case when the input is 1, 2, 3 for 1 clock cycle, then 16 for 3 clock cycle, then 24 for 3 clock cycle, and then 0 for 3 clock cycle. From the simulation waveform, we can see that when inputs are 1, 2, and 3, the expected output is just 0 since after they are divided by N and added together they are still less than 1. After 16 is inputed for 3 clock cycles, the output becomes 1, 2, 3 at each clock cycle. 16 divided by 16 is 1, they are accumulated each clock cycle so after 3 clock cycles the output should be 3. Similarly for 24 divided by 16 and added up. Lastly, 0 is inputed for the 3 clock cycle and it does not affect the output since 0 is just no value added to the accumulator. So, the above waveform matches our expected output.

Last, we use this filter to process noisy_left and noisy_right in the top module and output the results to task3_left and task3_right. Below in Fig. 12 is a screenshot of the synthesis report of task3.



Flow Summary

<<Filter>>

Flow Status	Successful - Mon May 24 06:17:20 2021
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	task1
Top-level Entity Name	task3
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	976
Total pins	76
Total virtual pins	0
Total block memory bits	12,288
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1
Total DLLs	0

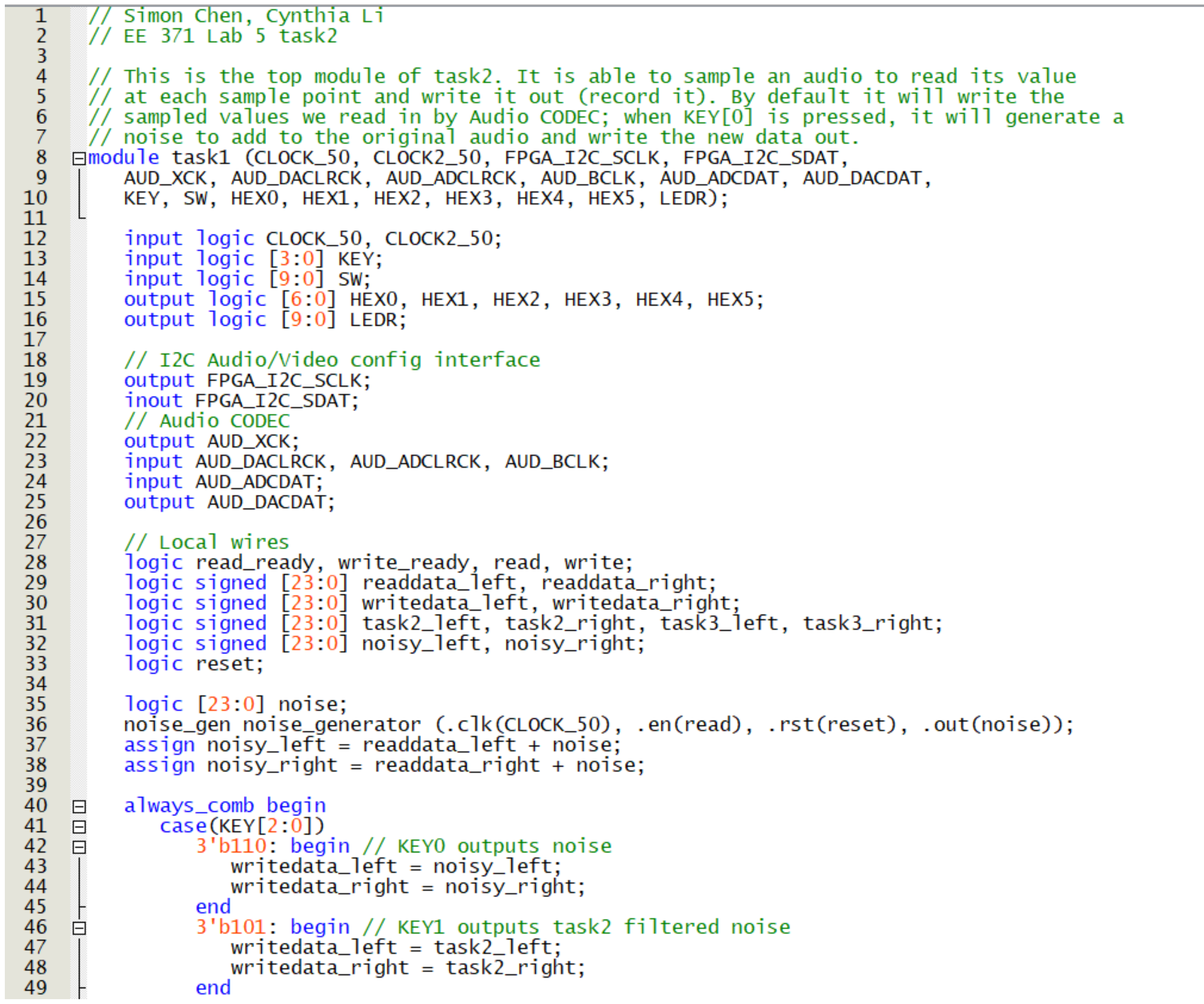
Fig. 12 Task 3 flow summary

Conclusion: After playing task 3 with different N, we found that the larger the N, the output sound should have smaller volume but have clearer sound than the filter in task 2. Our main takeaway for this lab is that we learned how to use the audio output and input on the DEI-SoC board to perform signal processing at the elementary level. We successfully implemented two different filters in task2 and task 3, but we can see their performances are not very good from the demo. Real life applications would be much more complicated. But overall, the output was as expected as the lab would desire.

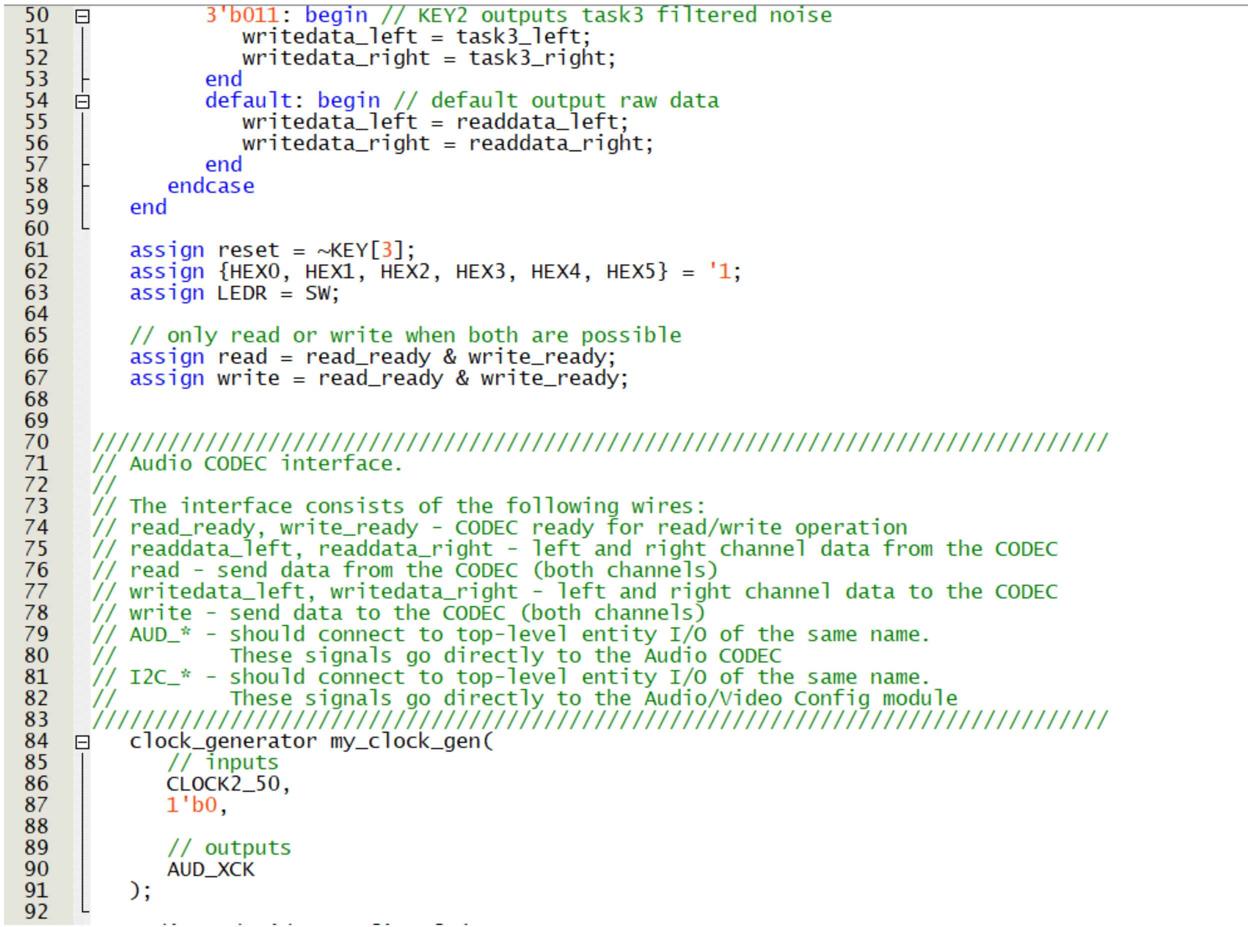
APPENDIX

I. Task 1:

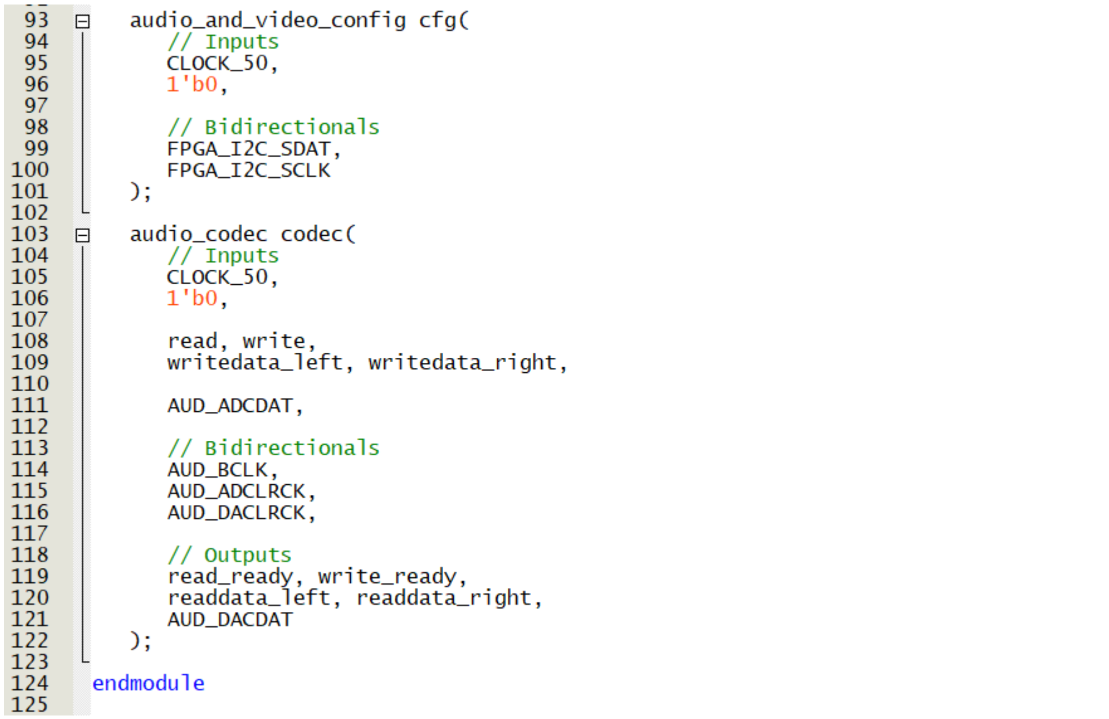
task1.sv



```
// Simon Chen, Cynthia Li
// EE 371 Lab 5 task2

// This is the top module of task2. It is able to sample an audio to read its value
// at each sample point and write it out (record it). By default it will write the
// sampled values we read in by Audio CODEC; when KEY[0] is pressed, it will generate a
// noise to add to the original audio and write the new data out.
module task1 (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 noise_generator (.clk(CLOCK_50), .en(read), .rst(reset), .out(noise));
   assign noisy_left = readdata_left + noise;
   assign noisy_right = readdata_right + noise;

   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
```

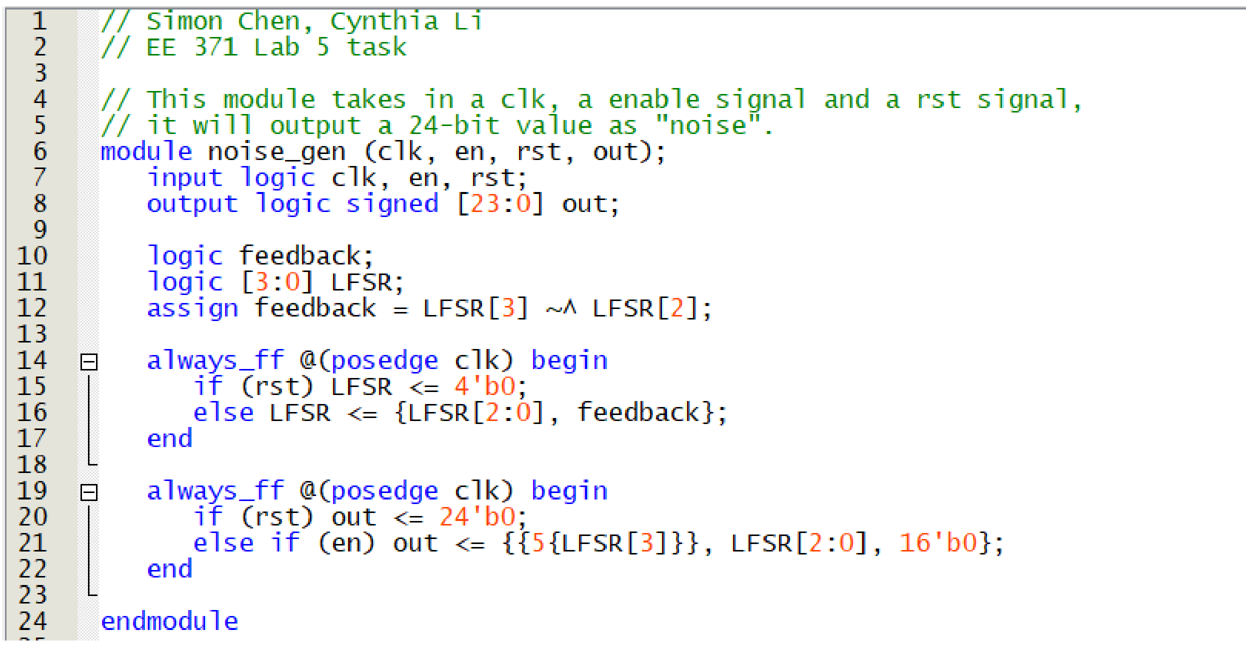


```
         3'b011: begin // KEY2 outputs task3 filtered noise
            writedata_left = task3_left;
            writedata_right = task3_right;
         end
         default: begin // default output raw data
            writedata_left = readdata_left;
            writedata_right = readdata_right;
         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;


/////////////////////////////////////////////////////////////////////////////////
// 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
   );

```

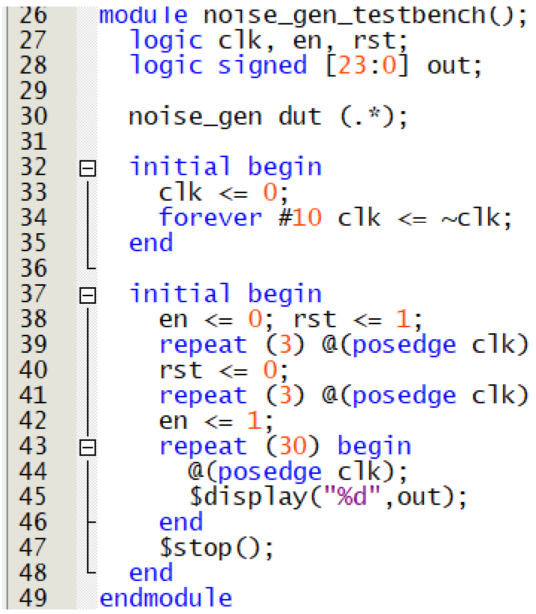


```
   audio_and_video_config cfg(
      // Inputs
      CLOCK_50,
      1'b0,

      // 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

```

noise_gen.sv



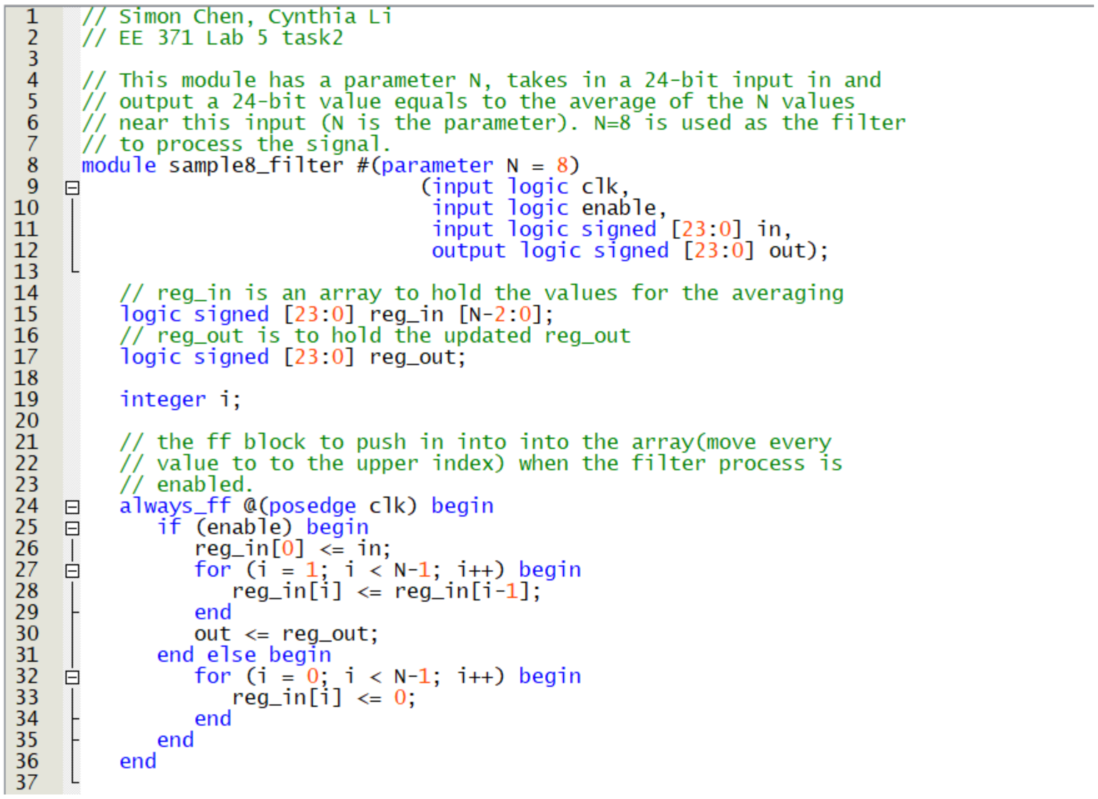
```systemverilog
// Simon Chen, Cynthia Li
// EE 371 Lab 5 task

// This module takes in a clk, a enable signal and a rst signal,
// it will output a 24-bit value as "noise".
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
```



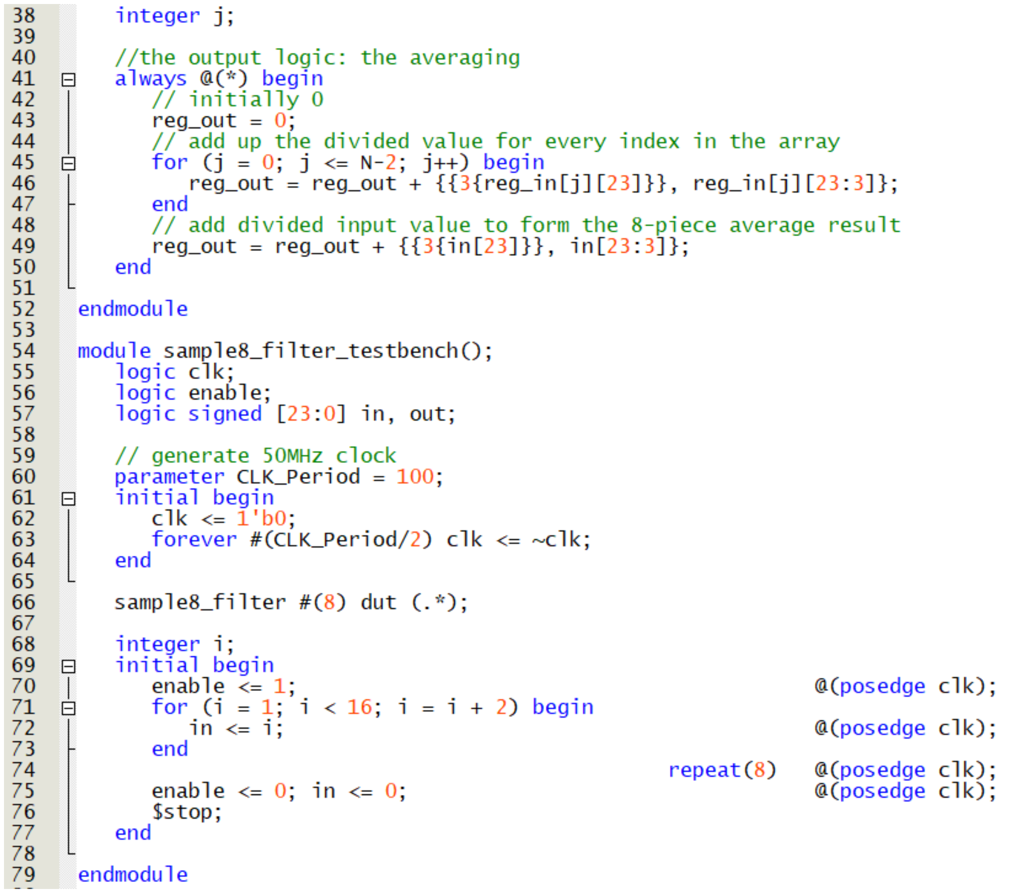
```systemverilog
module noise_gen_testbench();
  logic clk, en, rst;
  logic signed [23:0] out;

  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
```

II. Task 2

sample8_filter.sv

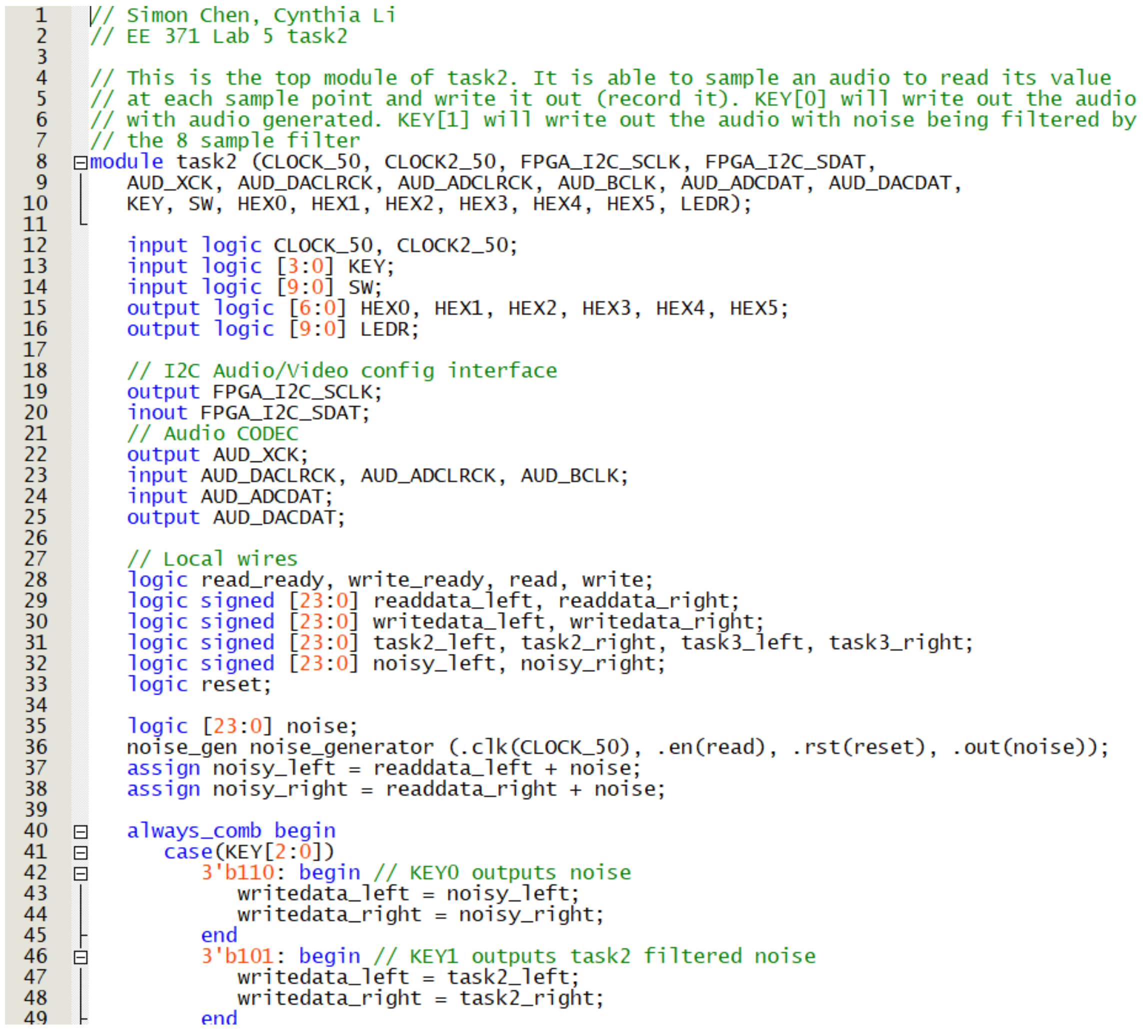


```
// Simon Chen, Cynthia Li
// EE 371 Lab 5 task2

// This module has a parameter N, takes in a 24-bit input in and
// output a 24-bit value equals to the average of the N values
// near this input (N is the parameter). N=8 is used as the filter
// to process the signal.
module sample8_filter #(parameter N = 8)
                           (input logic clk,
                            input logic enable,
                            input logic signed [23:0] in,
                            output logic signed [23:0] out);

   // reg_in is an array to hold the values for the averaging
   logic signed [23:0] reg_in [N-2:0];
   // reg_out is to hold the updated reg_out
   logic signed [23:0] reg_out;

   integer i;

   // the ff block to push in into into the array(move every
   // value to to the upper index) when the filter process is
   // enabled.
   always_ff @(posedge clk) begin
      if (enable) begin
         reg_in[0] <= in;
         for (i = 1; i < N-1; i++) begin
            reg_in[i] <= reg_in[i-1];
         end
         out <= reg_out;
      end else begin
         for (i = 0; i < N-1; i++) begin
            reg_in[i] <= 0;
         end
      end
   end
```

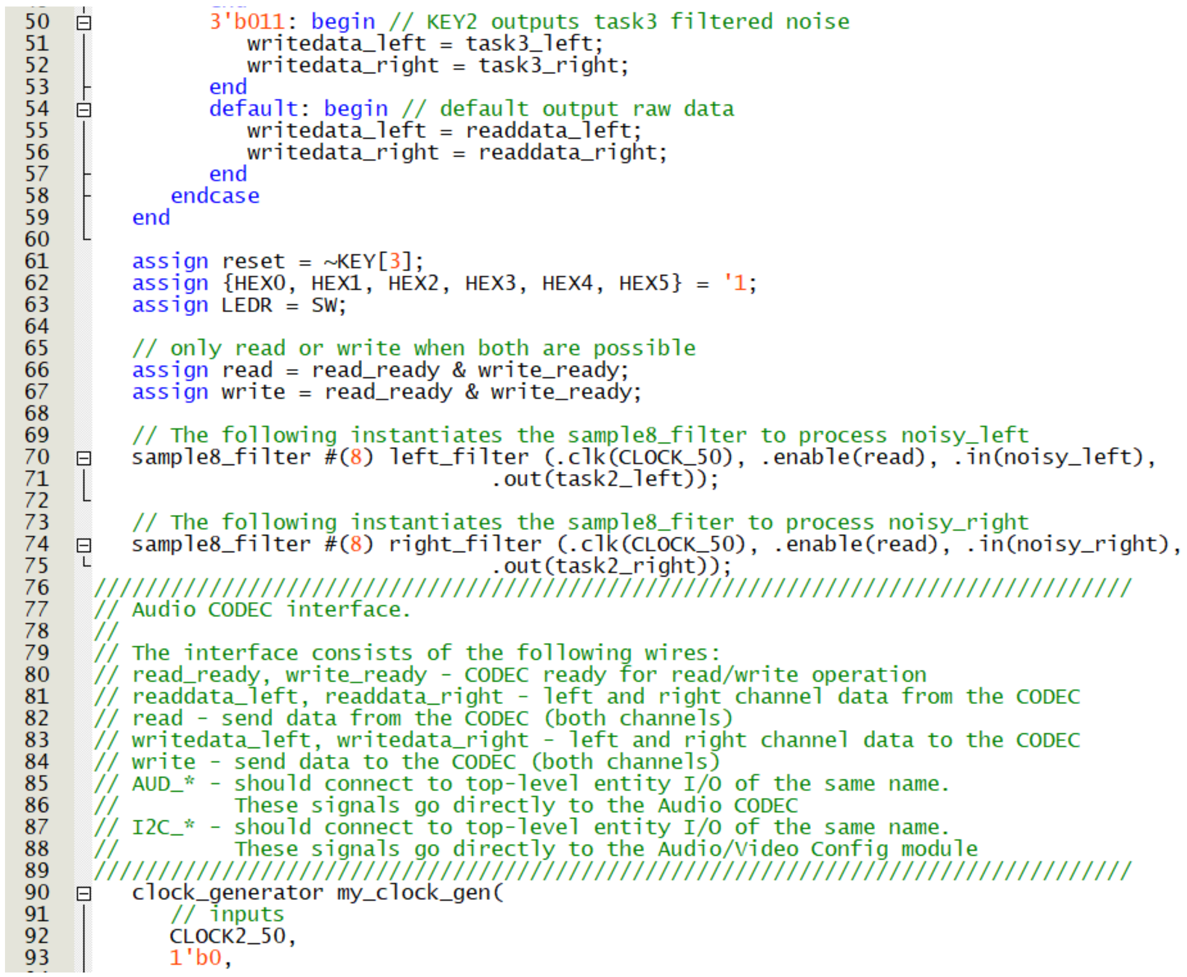


```
   integer j;

   //the output logic: the averaging
   always @(*) begin
      // initially 0
      reg_out = 0;
      // add up the divided value for every index in the array
      for (j = 0; j <= N-2; j++) begin
         reg_out = reg_out + {{3{reg_in[j][23]}}, reg_in[j][23:3]};
      end
      // add divided input value to form the 8-piece average result
      reg_out = reg_out + {{3{in[23]}}, in[23:3]};
   end

endmodule

module sample8_filter_testbench();
   logic clk;
   logic enable;
   logic signed [23:0] in, out;

   // generate 50MHz clock
   parameter CLK_Period = 100;
   initial begin
      clk <= 1'b0;
      forever #(CLK_Period/2) clk <= ~clk;
   end

   sample8_filter #(8) dut (.*);

   integer i;
   initial begin
      enable <= 1;                                           @(posedge clk);
      for (i = 1; i < 16; i = i + 2) begin
         in <= i;                                            @(posedge clk);
      end
                                                 repeat(8)   @(posedge clk);
      enable <= 0; in <= 0;                                  @(posedge clk);
      $stop;
   end

endmodule
```

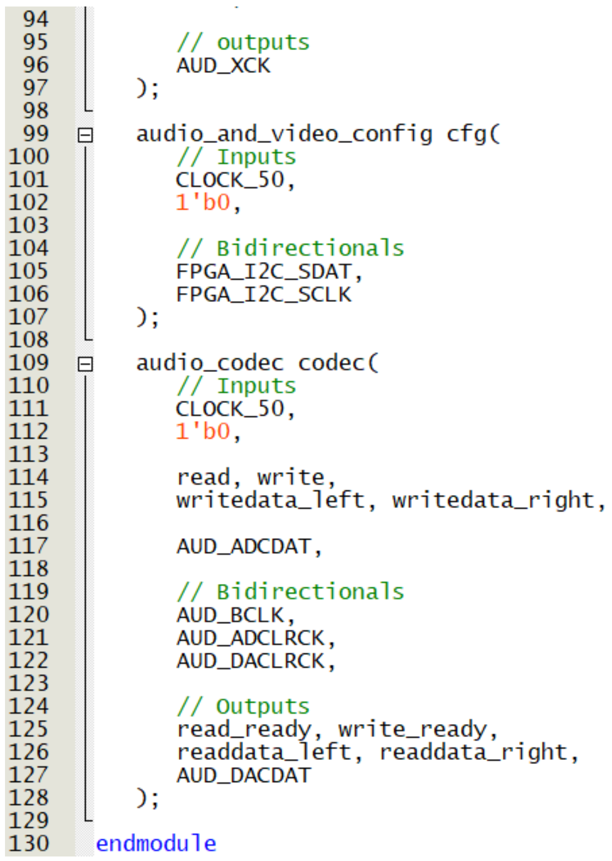
task2.sv



```systemverilog
// Simon Chen, Cynthia Li
// EE 371 Lab 5 task2

// This is the top module of task2. It is able to sample an audio to read its value
// at each sample point and write it out (record it). KEY[0] will write out the audio
// with audio generated. KEY[1] will write out the audio with noise being filtered by
// the 8 sample filter
module task2 (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 noise_generator (.clk(CLOCK_50), .en(read), .rst(reset), .out(noise));
   assign noisy_left = readdata_left + noise;
   assign noisy_right = readdata_right + noise;

   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
```



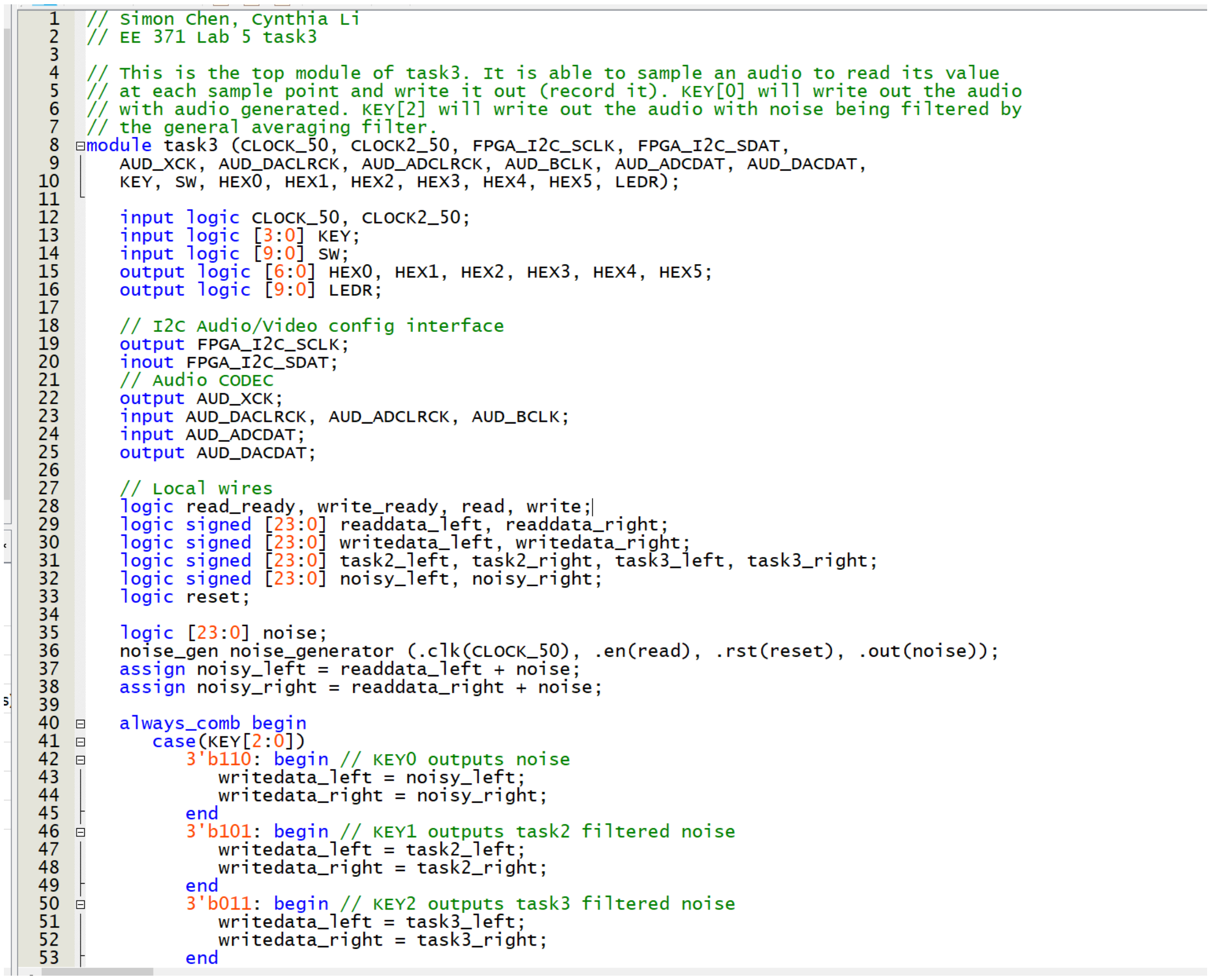
```
         3'b011: begin // KEY2 outputs task3 filtered noise
            writedata_left = task3_left;
            writedata_right = task3_right;
         end
         default: begin // default output raw data
            writedata_left = readdata_left;
            writedata_right = readdata_right;
         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;

   // The following instantiates the sample8_filter to process noisy_left
   sample8_filter #(8) left_filter (.clk(CLOCK_50), .enable(read), .in(noisy_left),
                               .out(task2_left));

   // The following instantiates the sample8_fiter to process noisy_right
   sample8_filter #(8) right_filter (.clk(CLOCK_50), .enable(read), .in(noisy_right),
                               .out(task2_right));
/////////////////////////////////////////////////////////////////////////////////
// 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,
```



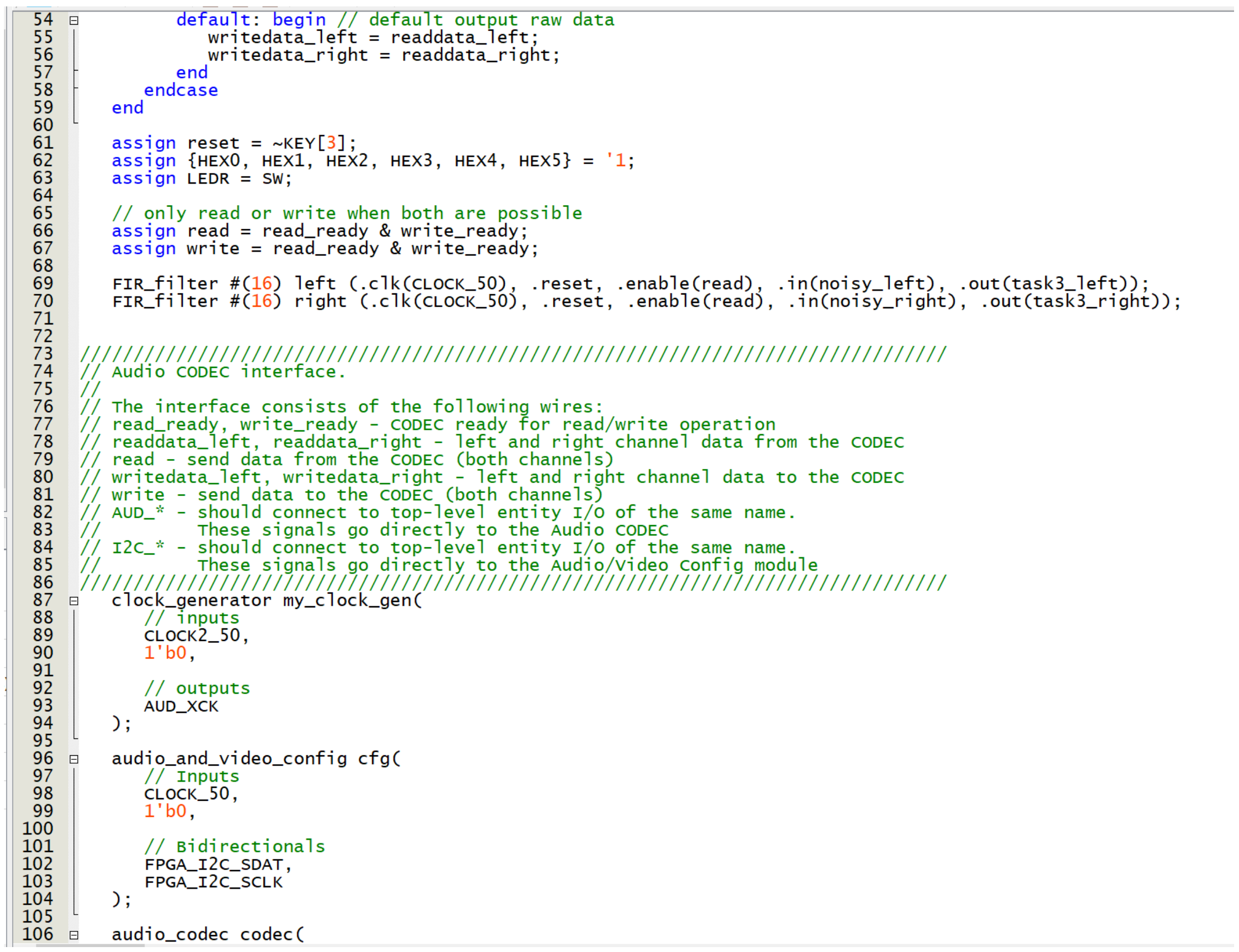
```
    
        // outputs
        AUD_XCK
    );

    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
```

III. Task 3

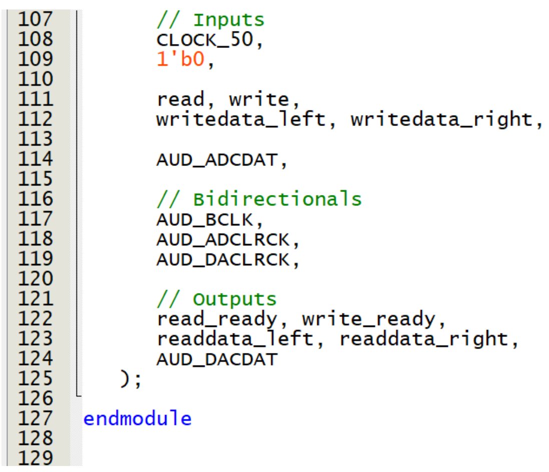
task3.sv



```
// Simon Chen, Cynthia Li
// EE 371 Lab 5 task3

// This is the top module of task3. It is able to sample an audio to read its value
// at each sample point and write it out (record it). KEY[0] will write out the audio
// with audio generated. KEY[2] will write out the audio with noise being filtered by
// the general averaging filter.
module task3 (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 noise_generator (.clk(CLOCK_50), .en(read), .rst(reset), .out(noise));
   assign noisy_left = readdata_left + noise;
   assign noisy_right = readdata_right + noise;

   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
```

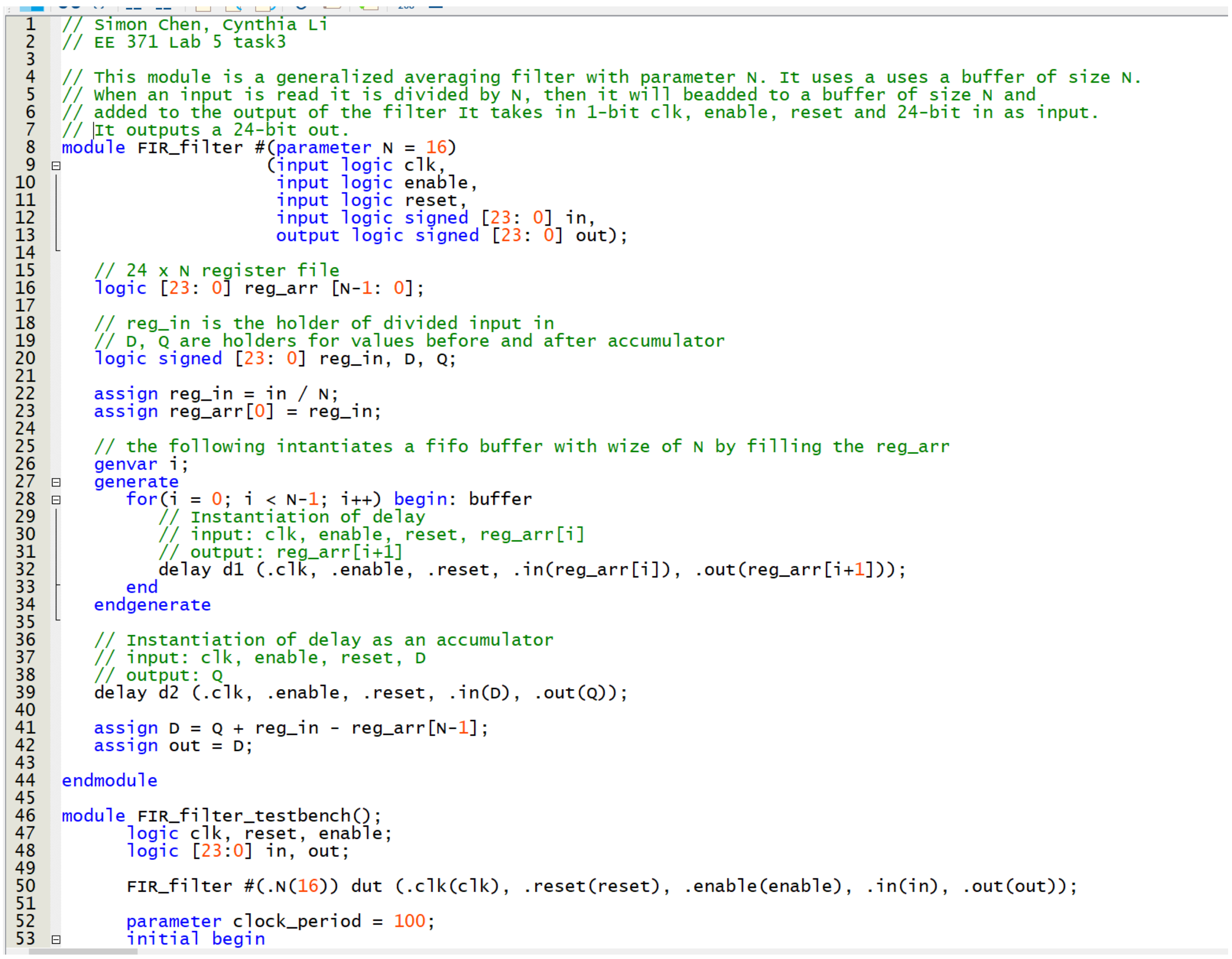


```
         default: begin // default output raw data
            writedata_left = readdata_left;
            writedata_right = readdata_right;
         end
      endcase
   end

   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;

   FIR_filter #(16) left (.clk(CLOCK_50), .reset, .enable(read), .in(noisy_left), .out(task3_left));
   FIR_filter #(16) right (.clk(CLOCK_50), .reset, .enable(read), .in(noisy_right), .out(task3_right));


/////////////////////////////////////////////////////////////////////////////////
// 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(
```

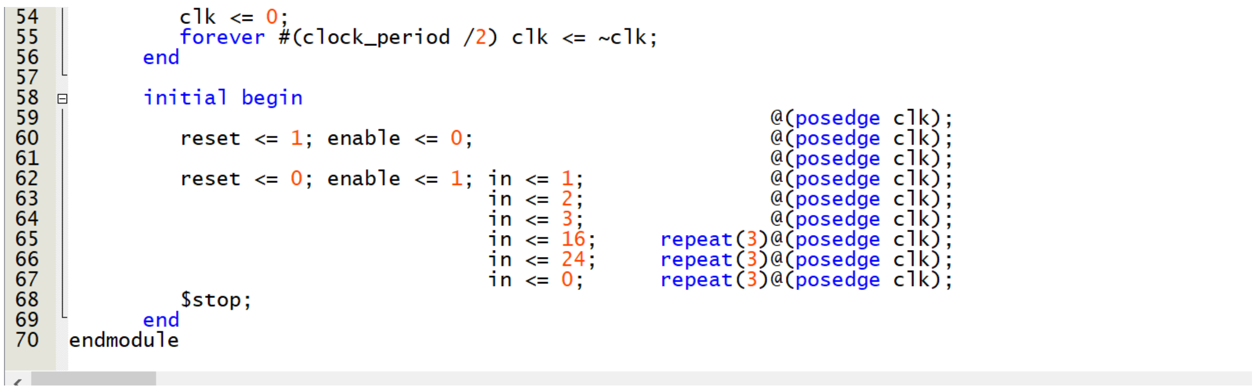


```
      // Inputs
      CLOCK_50,
      1'b0,

      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

```

FIR_filter.sv

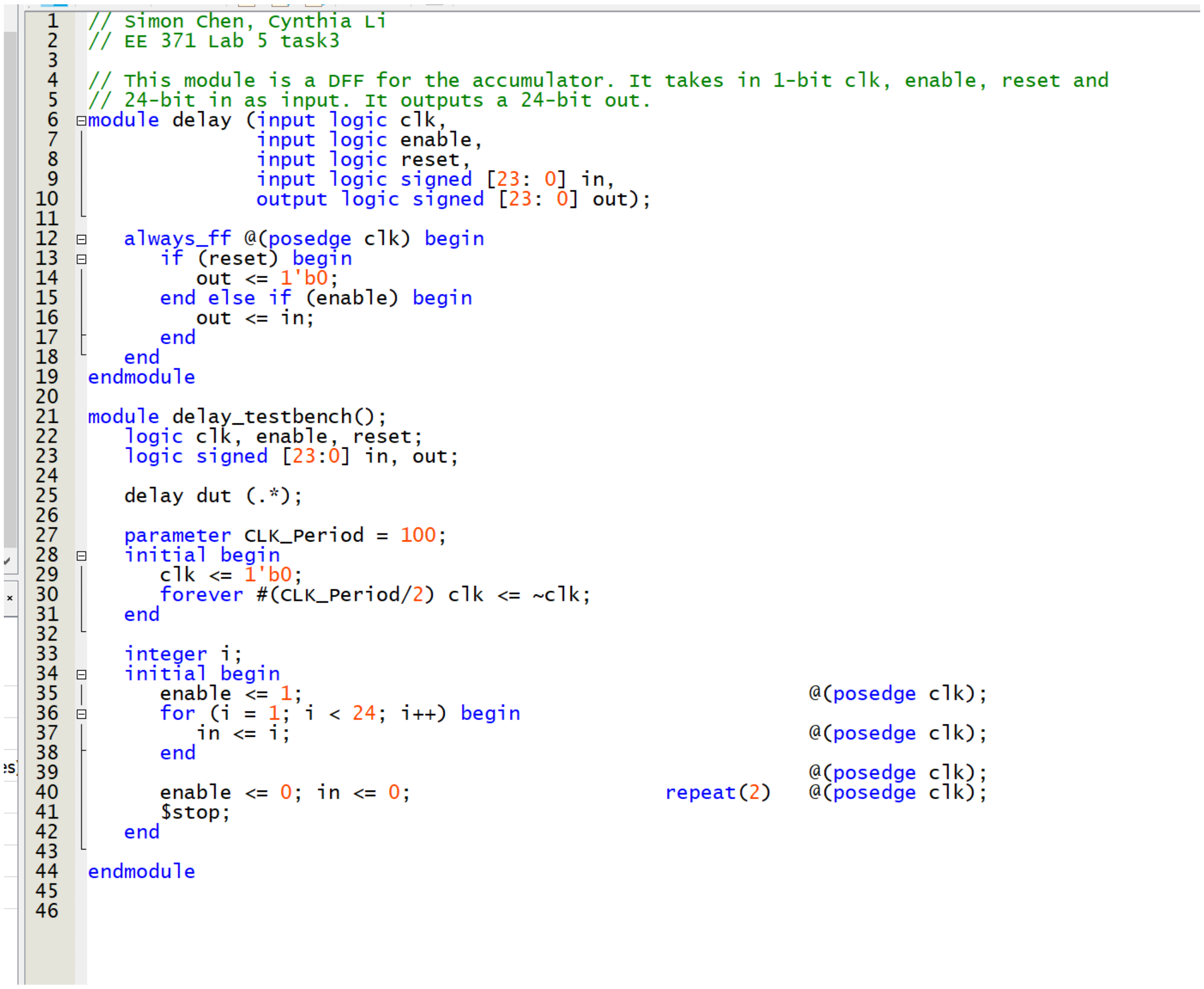


```
// Simon Chen, Cynthia Li
// EE 371 Lab 5 task3

// This module is a generalized averaging filter with parameter N. It uses a uses a buffer of size N.
// When an input is read it is divided by N, then it will beadded to a buffer of size N and
// added to the output of the filter It takes in 1-bit clk, enable, reset and 24-bit in as input.
// It outputs a 24-bit out.
module FIR_filter #(parameter N = 16)
                   (input logic clk,
                    input logic enable,
                    input logic reset,
                    input logic signed [23: 0] in,
                    output logic signed [23: 0] out);

   // 24 x N register file
   logic [23: 0] reg_arr [N-1: 0];

   // reg_in is the holder of divided input in
   // D, Q are holders for values before and after accumulator
   logic signed [23: 0] reg_in, D, Q;

   assign reg_in = in / N;
   assign reg_arr[0] = reg_in;

   // the following intantiates a fifo buffer with wize of N by filling the reg_arr
   genvar i;
   generate
      for(i = 0; i < N-1; i++) begin: buffer
         // Instantiation of delay
         // input: clk, enable, reset, reg_arr[i]
         // output: reg_arr[i+1]
         delay d1 (.clk, .enable, .reset, .in(reg_arr[i]), .out(reg_arr[i+1]));
      end
   endgenerate

   // Instantiation of delay as an accumulator
   // input: clk, enable, reset, D
   // output: Q
   delay d2 (.clk, .enable, .reset, .in(D), .out(Q));

   assign D = Q + reg_in - reg_arr[N-1];
   assign out = D;

endmodule

module FIR_filter_testbench();
      logic clk, reset, enable;
      logic [23:0] in, out;

      FIR_filter #(.N(16)) dut (.clk(clk), .reset(reset), .enable(enable), .in(in), .out(out));

      parameter clock_period = 100;
      initial begin
```



```
         clk <= 0;
         forever #(clock_period /2) clk <= ~clk;
      end

      initial begin
                                                            @(posedge clk);
         reset <= 1; enable <= 0;                           @(posedge clk);
                                                            @(posedge clk);
         reset <= 0; enable <= 1;   in <= 1;                @(posedge clk);
                                    in <= 2;                @(posedge clk);
                                    in <= 3;                @(posedge clk);
                                    in <= 16;      repeat(3)@(posedge clk);
                                    in <= 24;      repeat(3)@(posedge clk);
                                    in <= 0;       repeat(3)@(posedge clk);

         $stop;
      end
endmodule
```

delay.sv



```
// Simon Chen, Cynthia Li
// EE 371 Lab 5 task3

// This module is a DFF for the accumulator. It takes in 1-bit clk, enable, reset and
// 24-bit in as input. It outputs a 24-bit out.
module delay (input logic clk,
              input logic enable,
              input logic reset,
              input logic signed [23: 0] in,
              output logic signed [23: 0] out);

   always_ff @(posedge clk) begin
      if (reset) begin
         out <= 1'b0;
      end else if (enable) begin
         out <= in;
      end
   end
endmodule

module delay_testbench();
   logic clk, enable, reset;
   logic signed [23:0] in, out;

   delay dut (.*);

   parameter CLK_Period = 100;
   initial begin
      clk <= 1'b0;
      forever #(CLK_Period/2) clk <= ~clk;
   end

   integer i;
   initial begin
      enable <= 1;                                      @(posedge clk);
      for (i = 1; i < 24; i++) begin
         in <= i;                                       @(posedge clk);
      end
                                                        @(posedge clk);
      enable <= 0; in <= 0;                 repeat(2)   @(posedge clk);
      $stop;
   end

endmodule
```