Lab 5 Report: Audio Hardware and Noise Smoothing Filters

Hunter North and Peter Zhong

EE 371

May 17, 2021

Procedure

Task One

Task one did not ask us to modify any of the provided files. Instead, it simply asked us to load the design onto Labsland FPGA and try to record a noisy vs. not noisy audio output. To complete this task, we downloaded the provided files from Canvas, compiled the module, uploaded the module to Labsland, and used the interactive interface to play the piano sound into the FPGA and record the output audio. The simulation waveform for the noise generator used in task 1 and the result mp3 file that we have recorded for task 1 can be found below in the "Results" section of the report.

Task Two

Module: FIR_filter

Task two asks us to implement a simple averaging Finite Impulse Response (FIR) filter. This filter works by taking the average value of 8 adjacent samples and outputting the average value to Dataout (more information available in Figure 1).

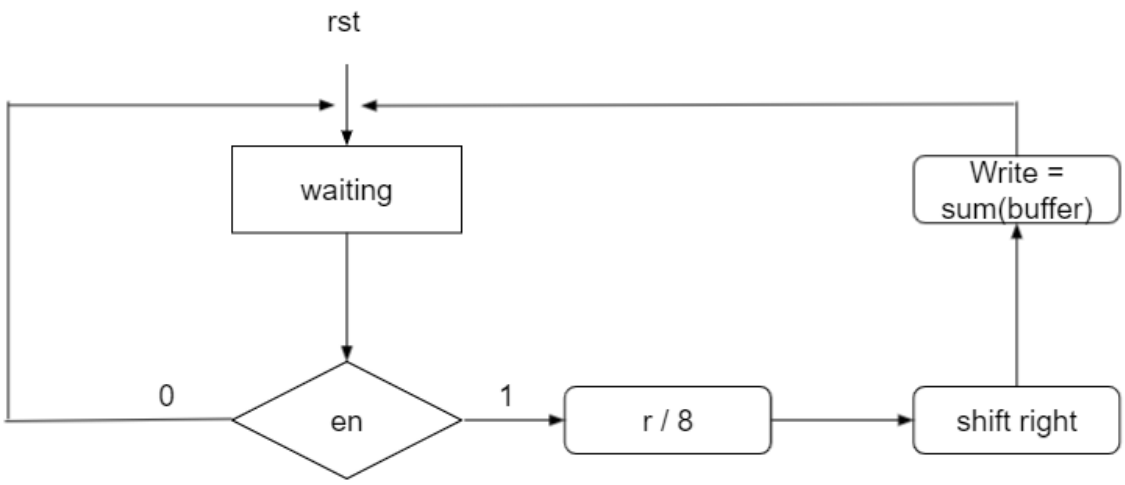

Figure 1. FIR_filter ASMD. Flowchart showing the process executed by our FIR_filter module.

In order to create this simple filter, we first used the formula provided by the lab spec to perform the division on the input samples by shifting data. Then, we created a buffer for both left and right channels to hold the values of the 8 input samples. Lastly, we calculated the average values by summing up the 8 values in the buffer and synchronously outputted that average value to the output ports when the module is enabled. Please refer to "FIR_filter.sv" for more details regarding our implementation of the simple FIR filter. To view the explicit input and output ports required to operate the module, see figure 2.

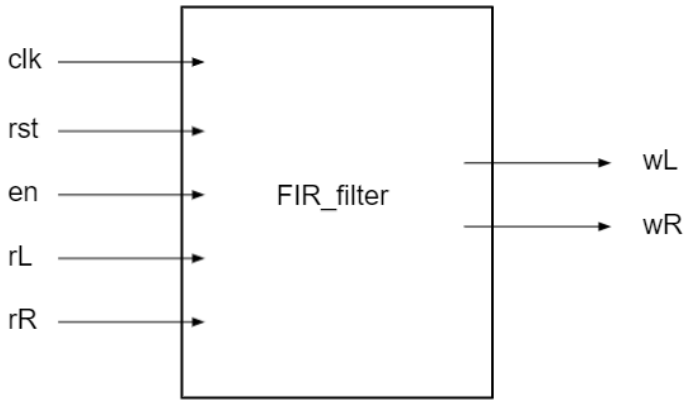

Figure 2. FIR_filter Block Diagram. Image showing the correct input and output signals required to drive the FIR_filter.

Task Three

Module: FIR_advanced_filter

Task three asks us to implement a more advanced FIR filter. On one hand, this advanced filter module is parameterized so that it can calculate the average value of any specified numbers of samples (N is the parameter). On the other hand, an additional accumulator is implemented in this module so that the output data is always the average value of the most recent N samples (more information available in Figure 1).

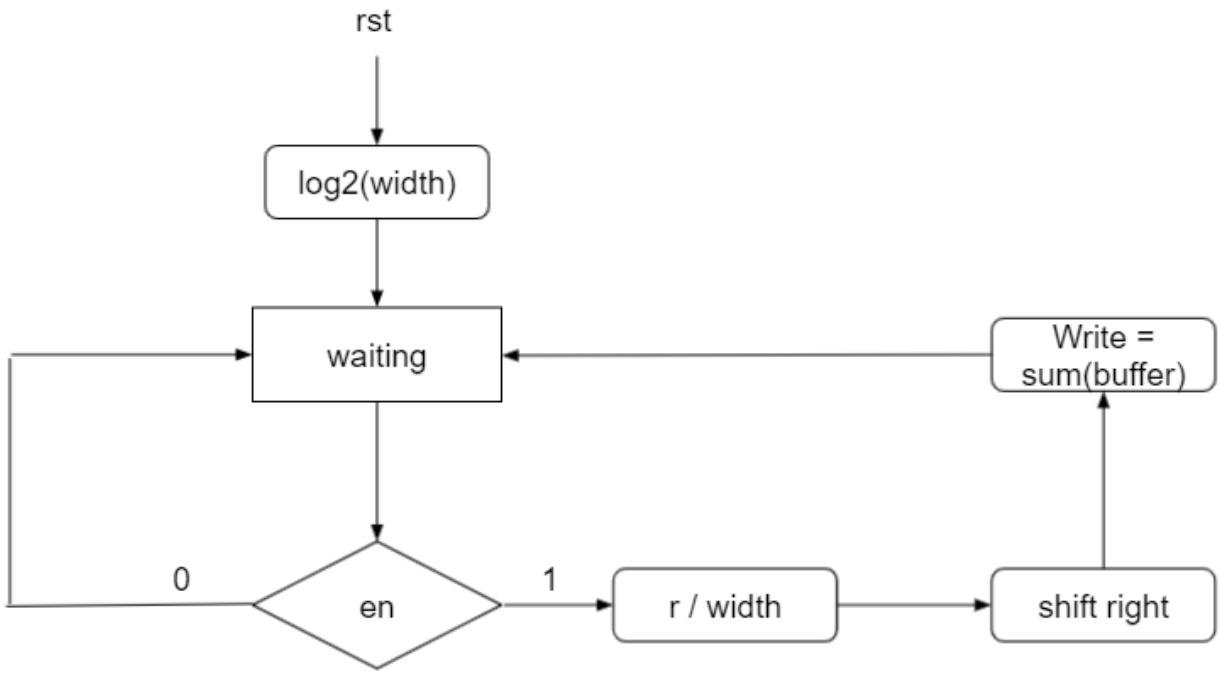

Figure 3. FIR_advanced_filter ASMD. Flowchart showing the process executed by our FIR_filter module.

In order to implement this more advanced filter, we first borrowed the code from the simple filter. Then, we parameterized the size of the buffer and introduced parameter n into other operations such as dividing and shifting buffer. Lastly, we used an always_ff block to implement the accumulator that removes the least recent data and adds the most recent data when the module is enabled. Please refer to "FIR_advanced_filter.sv" for more details regarding our implementation of the simple FIR filter. To view the explicit input and output ports required to operate the module, see figure 4.

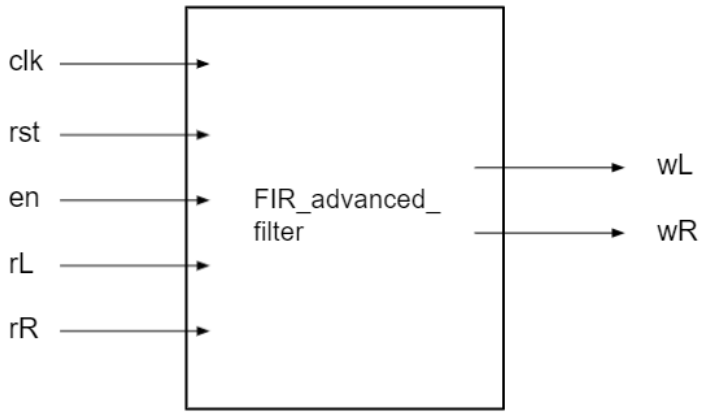

Figure 4. FIR_advanced_filter Block Diagram. Image showing the correct input and output signals required to drive the FIR_advanced_filter.

Results

Task One - Please refer to task1.mp3 for audio demonstration

Module: noise_gen

This waveform below (Figure 5) shows the successful testing of the noise_gen module. The noise_gen module is a random 24-bit number generator with the LFSR design. To test the module, we simply resetted the module and let it run for some clock cycles. As can be seen from the waveform below, the noise_gen module outputs generally random numbers to the output port over different clock cycles.

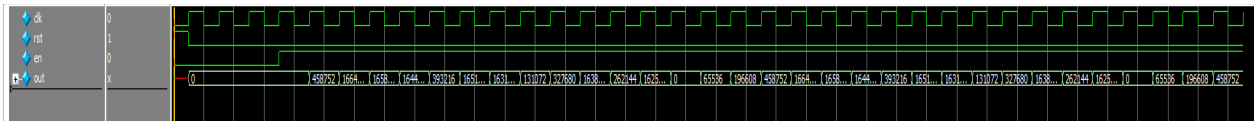

Figure 5. Waveform generated by Appendix 1.B that demonstrates a successful test of the noise_gen module.

Task Two- Please refer to task2.mp3 for audio demonstration

This waveform below (Figure 6) shows the successful testing of the FIR_filter module. The module takes the average value of 8 adjacent samples from both rR and rL ports, and outputs the average value to the wL and wR ports.

To test this module, we resetted the entire module, loaded 8, 16, and 0 into rR, and loaded 16, 32, and 0 into rL over 18 clock cycles. As can be seen from the waveform below, the output values first gradually increase to the input values when the input values are initially being loaded in, and the output values also dynamically adjust when different input values are being loaded into the module.

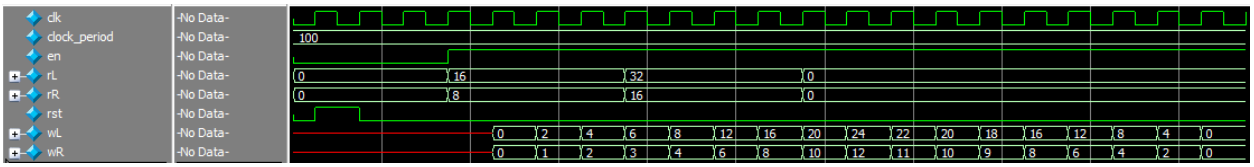

Figure 6. Waveform generated by Appendix 2.A that demonstrates a successful test of the FIR_filter module.

Task Three - Please refer to task3.mp3 for audio demonstration

This waveform below (Figure 7) shows the successful testing of the FIR_advanced_filter module. The module takes the average value of 16 adjacent samples (N=16 by default) from both rR and rL ports, and outputs the average value to the wL and wR ports. This

module also ensures that the output value is equal to the average value of the most recent 16 samples.

To test this module, we resetted the entire module, loaded 16, 32, 64, and 128 into rR, and loaded 32, 64, 128, and 256 into rL over 20 clock cycles. The behavior of this module is similar to the FIR_filter module in task 2. As can be seen from the waveform below, the output values first gradually increase to the input values when the values are initially being loaded in, and the output values also dynamically adjust when different input values are being loaded into the module.

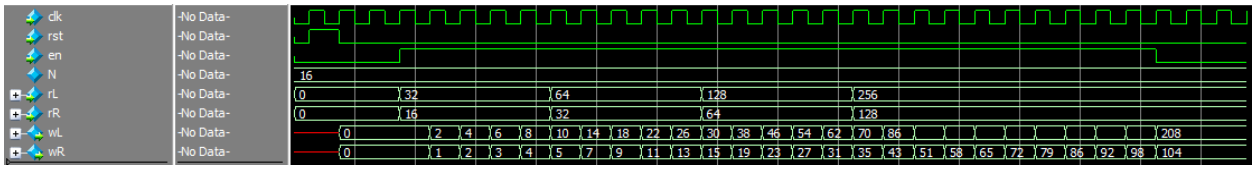

Figure 7. Waveform generated by Appendix 3.A that demonstrates a successful test of the FIR_advanced_filter module.

Appendix

1.A - DE1_SoC.sv

```
// Hunter North and Peter Zhong
// 05/17/2021
// EE 371
// Lab 5: Task 1

// DE1_SoC instantiates the noise generator, the simple FIR filter, and the advanced FIR filter.
// It establishes connections so that no key presses outputs raw data, KEY0 outputs noise, KEY1
// outputs task2 filtered noise, and KEY2 outputs task3 filtered noise.

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;

   // Instantiates noise generator for KEY0
   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;

   // Instansiates FIR_filter from task 2 for KEY1
   FIR_filter t2 (.clk(CLOCK_50), .rst(reset), .en(read), .rL(noisy_left), .rR(noisy_right),
                  .wL(task2_left), .wR(task2_right));

   // Instansiates FIR_advanced_filter from task 3 for KEY2
   FIR_advanced_filter t3 (.clk(CLOCK_50), .rst(reset), .en(read), .rL(noisy_left), .rR(noisy_right),
                           .wL(task3_left), .wR(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;
```

```
//////////////////////////////////////////////////////////////////////////////
// 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
```

```
module DE1_SoC_testbench ();

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

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

    // Insantiate the module for testing
    DE1_SoC dut (.*);

    initial begin
        // Set default values
        KEY[3] <= 1; KEY[2] <= 1;  KEY[1] <= 1; KEY[1] <= 0;              @(posedge CLOCK_50);

        // Reset
        KEY[3] <= 0;                                                      @(posedge CLOCK_50);
        KEY[3] <= 1;                                                      @(posedge CLOCK_50);

        // Listen to song normally
        KEY[3] <= 1;                                          repeat(50)  @(posedge CLOCK_50);

        // Add noise
        KEY[0] <= 0;                                          repeat(50)  @(posedge CLOCK_50);

        // Apply simple filter
        KEY[0] <= 1; KEY[1] <= 0;                             repeat(50)  @(posedge CLOCK_50);

        // Apply better filter
        KEY[1] <= 1; KEY[2] <= 0;                             repeat(50)  @(posedge CLOCK_50);

        $stop;
    end
endmodule
```

1.B - noise_gen.sv

```
// Hunter North and Peter Zhong
// 05/17/2021
// EE 371
// Lab 5: Task 1

// noise_gen is an LFSR-based 14-bit random number generator. It serves as a noise source
// for playing audio in the top level module.

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

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.A - FIR_filter.sv

```
// Hunter North and Peter Zhong
// 05/17/2021
// EE 371
// Lab 5: Task 2

// FIR_filter is an averaging filter that removes noise from an audio sample by
// taking the average of the adjacent 8 samples and outputting the average value
// to the output terminal.

module FIR_filter(clk, rst, en, rL, rR, wL, wR);

   input logic clk, rst, en;
   input logic [23:0] rR, rL;
   output logic [23:0] wR, wL;

   // Establish unit to govern shifting of 8 previous segments
   logic [23:0] bufferR [7:0];
   logic [23:0] bufferL [7:0];
   integer k;
   always_ff @(posedge clk) begin
      // reset to zeros
      if (rst) begin
         for (k = 7; k >= 0; k--) begin
            bufferR[k] <= '0;
            bufferL[k] <= '0;
         end

      // Update buffer - shift in the input divided by eight
      end else if (en) begin
         bufferR[7] <= {{3{rR[23]}}, rR[23:3]};
         bufferL[7] <= {{3{rL[23]}}, rL[23:3]};

         for (k = 6; k >= 0; k--) begin
            bufferR[k] <= bufferR[k+1];
            bufferL[k] <= bufferL[k+1];
         end
      end
   end

   // Perform summing operation
   logic [23:0] avR, avL;
   assign avR =   (bufferR[7] + bufferR[6] + bufferR[5] + bufferR[4] +
                   bufferR[3] + bufferR[2] + bufferR[1] + bufferR[0]);
   assign avL =   (bufferL[7] + bufferL[6] + bufferL[5] + bufferL[4] +
                   bufferL[3] + bufferL[2] + bufferL[1] + bufferL[0]);

   // Assign averages to the output
   always_ff @(posedge clk) begin
      if (en) begin
         wL <= avL;
         wR <= avR;
      end
   end
endmodule
```

```
// Testbench for the FIR_filter module
module FIR_filter_testbench();

   logic clk, rst, en;
   logic [23:0] rR, rL;
   logic [23:0] wR, wL;

   // Simulated clock for the testing
   parameter clock_period = 100;
   initial begin
      clk <= 0;
      forever #(clock_period / 2) clk <= ~clk;
   end

   // Insantiate the module for testing
   FIR_filter dut (.*);

   initial begin
      rst <= 0; en <= 0; rR <= 0; rL <= 0;                    @(posedge clk);

      // Reset system
      rst <= 1;                                               @(posedge clk);
      rst <= 0;                                         repeat(2) @(posedge clk);

      // Load in values
      en <= 1; rR <= 8;  rL <= 16;                      repeat(4) @(posedge clk);
      en <= 1; rR <= 16; rL <= 32;                      repeat(4) @(posedge clk);
      en <= 1; rR <= 0;  rL <= 0;                       repeat(10) @(posedge clk);
      $stop;
   end
endmodule
```

3.A - FIR_advanced_filter.sv

```
// Hunter North and Peter Zhong
// 05/17/2021
// EE 371
// Lab 5: Task 3

// FIR_advanced filter is a parametrized filter that dynamically takes the average value of
// the most recent N samples of the input data.

module FIR_advanced_filter #(parameter [7:0] N = 16)
                            (clk, rst, en, rL, rR, wL, wR);

   input logic clk, rst, en;
   input logic [23:0] rR, rL;

   output logic [23:0] wR, wL;

   // Place log2(N) into n with ceiling operation
   logic [7:0] n;
   always_comb begin
      if       (N < 2)     n = 0;
      else if  (N < 4)     n = 1;
      else if  (N < 8)     n = 2;
      else if  (N < 16)    n = 3;
      else if  (N < 32)    n = 4;
      else if  (N < 64)    n = 5;
      else if  (N < 128)   n = 6;
      else if  (N < 256)   n = 7;
   end

   // Divide input by window size
   logic [7:0] rRDiv, rLDiv;
   assign rRDiv = rR >> n;
   assign rLDiv = rL >> n;

               // Establish unit to govern shifting of N previous segments
               logic [23:0] bufferR [N-1:0];
               logic [23:0] bufferL [N-1:0];
               logic [23:0] outR, outL;
               integer k;
               always_ff @(posedge clk) begin
                  // reset to zeros
                  if (rst) begin
                     for (k = N-1; k >= 0; k--) begin
                        bufferR[k] <= '0;
                        bufferL[k] <= '0;
                     end
                     outR <= '0;
                     outL <= '0;
                  // Update buffer - shift in the input divided by N
                  end else if (en) begin
                     // Shift in
                     bufferR[N-1] <= rRDiv;
                     bufferL[N-1] <= rLDiv;

                     // Move data down
                     for (k = N-2; k >= 0; k--) begin
                        bufferR[k] <= bufferR[k+1];
                        bufferL[k] <= bufferL[k+1];
                     end

                     // Shift out
                     outR <= bufferR[0];
                     outL <= bufferL[0];
                  end
               end

               // Manage accumulation
               logic [23:0] accR, accL;
               always_ff @(posedge clk) begin
                  // Reset accumulators to zero
                  if (rst) begin
                     accR <= 0;
                     accL <= 0;
                  // Update accumulation
                  end else if (en) begin
                     accR <= (accR + rRDiv - outR);
                     accL <= (accL + rLDiv - outL);
                  end
               end

               assign wL = accL;
               assign wR = accR;
            endmodule
```

```
// Testbench for the FIR_advanced_filter module
module FIR_advanced_filter_testbench();

   logic clk, rst, en;
   logic [23:0] rR, rL;
   logic [23:0] wR, wL;

   // Simulated clock for the testing
   parameter clock_period = 100;
   initial begin
      clk <= 0;
      forever #(clock_period / 2) clk <= ~clk;
   end

   // Insantiate the module for testing
   FIR_advanced_filter #(16) dut (.*);

   initial begin
      rst <= 0; en <= 0; rR <= 0; rL <= 0;                    @(posedge clk);

      // Reset system
      rst <= 1;                                               @(posedge clk);
      rst <= 0;                                    repeat(2) @(posedge clk);

      // Load in values
      en <= 1; rR <= 16;  rL <= 32;                repeat(5) @(posedge clk);
      en <= 1; rR <= 32;  rL <= 64;                repeat(5) @(posedge clk);
      en <= 1; rR <= 64;  rL <= 128;               repeat(5) @(posedge clk);
      en <= 1; rR <= 128; rL <= 256;               repeat(10)@(posedge clk);

      en <= 0;                                     repeat(3) @(posedge clk);

      $stop;
   end
endmodule
```