****

**EE314 – Mini-Project**

**Task B: Filter & Decimator**

**Group 8**

**GROUP: Group B DATE: 21/04/2023**

*I declare that I have not plagiarised or excessively quoted the work of others, nor have I colluded*

*with others to represent collaborative work as my own. I have appropriately cited all information*

*derived from the published and unpublished work of other and less than 5% of my submission*

*quotes the source material directly.*

**NAME: Patrick Kavanagh SIGNED: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**NAME: Alessandro Lupo SIGNED: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**NAME: James Florin Petri SIGNED: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Project Completion Plan:

|  |  |  |
| --- | --- | --- |
| Tasks | Description | Name |
| Research | The functionality of a decimator and the best way of implementing the decimator where researched. As well as how the decimator is used in industry in conjunction with filters. | Alessandro Lupo  Patrick Kavanagh  James Petri |
| Module Coding | A module was created to implement the functionality of the filter and of the decimator. The first step to achieve this was to implement the second order low pass function. Secondly the average of the last 64 outputs of the filter results. The last step is to down sample the data by selecting one sample every 64 samples. | Alessandro Lupo  Patrick Kavanagh  James Petri |
| Testbench Coding | The testbench was created to test the full functionality of the module. Different data streams where passed though the module to test a broad range of values to show the ability of the module to work with different values. Edge cases where also tested. All the possible data streams couldn’t be tested for as it would take to much time. | Alessandro Lupo  Patrick Kavanagh  James Petri |

# Introduction (~ 1 page):

## Block Diagram

DUT

in

IN

8

Decimator

out

rst

OUT

RST

CLK

Filter.v

Filter\_TB.v

clk

## Description of the use of such a module in everyday systems

Decimation is a signal processing technique that is commonly used in a wide range of applications. Its primary purpose is to reduce the sampling rate of a signal while retaining its essential information. This is achieved by eliminating the unnecessary data points from the signal that fall outside of the frequency range of interest, resulting in a lower sampling frequency.

The technique is particularly useful when the Nyquist frequency of a signal is much higher than the highest frequency component present in the signal. By removing these extraneous data points, decimation can significantly reduce the amount of data that needs to be processed, thereby decreasing the computational power required to analyse the signal. Additionally, decimators can improve the signal to noise ratio by filtering out unwanted noise present in the signal.

Another advantage of decimation is its ability to save memory or storage space. This can be critical in applications that involve large amounts of data, such as audio or video processing, where reducing the sampling rate of the signal can significantly decrease the amount of storage required. Furthermore, decimation can also help to mitigate aliasing effects that may occur when the signal is under sampled, resulting in a more accurate representation of the original signal. Overall, decimation is a powerful tool in signal processing that can improve the efficiency and accuracy of a wide range of applications.

# Verilog Testbench (with Comments):

// Testbench code for Filter & Decimator

module Filter\_TB();

  reg clk;

  reg rst;

  reg in;

  reg [127:0] dataword;

  wire [7:0]out;

  Filter DUT(

    .CLK(clk),

    .RST(rst),

    .IN(in),

    .OUT(out)

  );

  initial begin

    $dumpfile("dump.vcd");

    $dumpvars(2);

    $monitor("out=%d", out);

    clk = 1'b0;

    in = 1'b0;

    dataword = 0;

    //reset at start to set values

    rst = 1'b1;

    #100

    rst <= 1'b0;

    #100

    rst <= 1'b1;

    // data upload 1    xxV0123456789ABC?

    dataword = 128'h00000000000000000000000000000000;

    #6425 //edge case ^

    dataword = 128'h02FF37abc326a7202381f0FF2a23f6be;

    #6425

    dataword = 128'h3864292ae9600269F0FF73abc3b7a721;

    #6425

    dataword = 128'h02395830aFF2a23FFbe3864792ae9603;

    #6425

    dataword = 128'h312899abc02138698329aef923073785;

    #6425

    dataword = 128'h0239830aFF2a23698329aef923073785;

    #6425

    dataword = 128'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;

    #6425 //edge case ^

   $finish;

  end

  always #25

  begin

    clk <= ~clk;

  end

  always @(posedge clk) begin

    in <= dataword[127];

    dataword <= { dataword[126:0], 1'b0 }; //left wise bit shifts

    //$display ("in = %b", in);

  end

endmodule

# Verilog Module(s) (with Comments):

//filter and decimator

module Filter(input CLK,

              input RST,

              input IN,

              output reg [7:0] OUT);

  // Declaring the filter parameters

  parameter a0 = 1;

  parameter a1 = -2;

  parameter a2 = 1;

  // Declaring the circular averaging buffer

  parameter BUF\_SIZE = 64;

  reg [7:0] buffer [0:BUF\_SIZE-1];

  reg [6:0] buf\_counter;

  // Downsampling counter

  reg [6:0] ds\_counter;

  // Filter state variables

  reg [7:0] x;

  reg [7:0] y1;

  reg [7:0] y2;

  // Integer used for loops

  integer i;

  reg signed [9:0] sum;

  reg [7:0] avg;

  always @ (posedge CLK or negedge RST) begin

    if(RST == 0) begin

      // Reset output to 0

      OUT <= 0;

      // Reset the filter state variables

      y1 <= 0;

      y2 <= 0;

      // Reset the averaging buffer

      buf\_counter <= 0;

      for (i = 0; i < BUF\_SIZE; i=i+1) begin

        buffer[i] = 0;

      end

      // Reset the downsample counter

      ds\_counter <= 0;

    end

    else begin

      // Filter the input sample and add it to the buffer. The ternary operator

      // prevents rollover.

      x <= IN;

      buffer[buf\_counter] = (IN + y2 > 2 \* y1) ? a0 \* IN + a1 \* y1 + a2 \* y2 : 0;

      // Update the filter state variables

      y2 <= y1;

      y1 <= x;

      // Increase the counter and reset to 0 if it gets to 64

      buf\_counter <= (buf\_counter == BUF\_SIZE - 1) ? 0 : buf\_counter + 1;

      // Compute the running average of the buffer

      sum = 0;

      for (i = 0; i < BUF\_SIZE; i = i + 1) begin

        sum = sum + buffer[i];

      end

      // Getting the average of the buffer values. The ternary prevents

      // the flooring when dividing, by rounding to the closest int

      // between 0 and 1

      avg <= ((sum \* 10) / BUF\_SIZE < 5) ? 0 : 1;

      // Downsample output by selecting one in every 64 samples

      ds\_counter <= ds\_counter + 1;

      if (ds\_counter == 64) begin

        OUT <= avg;

        ds\_counter <= 0;

      end else begin

        OUT <= 0;

      end

    end

  end

endmodule

# Description of the Design

The best way to describe the functionality of the design is to break it up into steps.

Filtering with a second-order low pass function: The input data stream is first passed through a second-order low pass filter with a transfer function of . This filter is designed to remove high-frequency noise and other unwanted signals from the audio data.

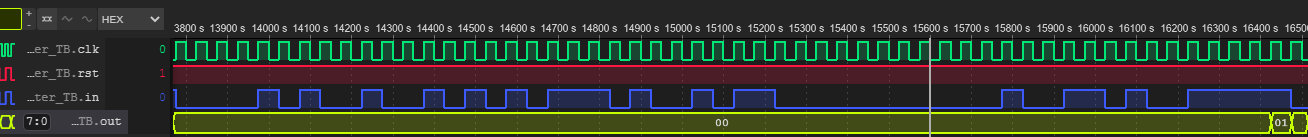
Running average calculation: The output of the filter is then used to calculate a running average of the past 64 filter outputs. This step helps to smooth out any remaining noise or irregularities in the signal.

Down-sampling to 44 kHz: The processed data is then down-sampled by selecting one sample out of every 64 samples. This step reduces the data rate and helps to further simplify the signal processing chain.

The final step involves generating an 8-bit output signal. The output signal is represented using 8 bits, which allows for a wide range of amplitude levels.

# Simulated Results

Indicate examples or sections of signals from a testbench simulation which clearly demonstrate the module(s) work as expected.



A screenshot of a computer

Description automatically generated with low confidence

A screenshot of a computer

Description automatically generated with low confidence The average changes between 1 and 0 as the average is always less than 64, it would be floored to 0. As Verilog doesn’t support floating point numbers, to circumvent this, we needed to add in a piece of code to check how the number would traditionally be rounded (<0.5 =0 , >0.5=1), on the left shows how it would look like in practice with the picture on the right showing the average numbers if they were multiplied by 10, where the out on the right is less than 5 the out on the left is 0 whereas if the output is 5 or greater on the right it will be 1 on the left (in practice).

# Conclusions

## Explain in words how you validated that the design works correctly, and how it is possible to tell that it works.

Manual calculations were performed with the given inputs and checked to see if the outputs matched.

## What additional features if any did you incorporate into the design or testbench to improve it?

Edge cases were added to the testbench.

## What are the limitations of your design?

The input needs to be 1 bit, so larger inputs cannot be passed through at once. It can only take in one data stream at a time