EE5516 : VLSI Architectures for Signal Processing and Machine Learning

Implementation of LMS (Least Mean Square) and RLS (Recursive Least Square) Adaptive Wiener Filters and Comparison of their Convergence

Nakul C (122101024) and Ashwin R Nair (12210104)

Abstract

This course project implements a hardware design for a 4-tap Wiener filter using both Least Mean Squares (LMS) and Recursive Least Squares (RLS) algorithms. The design is realized in Verilog Hardware Description Language (HDL). The project focuses on comparing the convergence behavior of these two adaptive filter algorithms.

1 Introduction

Adaptive filtering is a critical technique in signal processing, used to tailor filters to changing environments and varying signal conditions. Among the various adaptive filtering algorithms, the Least Mean Square (LMS) and Recursive Least Squares (RLS) algorithms stand out due to their widespread applications and distinct characteristics.

The LMS algorithm, developed by Widrow and Hoff in 1960, is renowned for its simplicity and ease of implementation. It operates on the principle of minimizing the mean square error between the desired signal and the filter output by adjusting the filter coefficients iteratively. Despite its computational efficiency, the LMS algorithm typically exhibits slower convergence, which can be a limitation in applications requiring rapid adaptation.

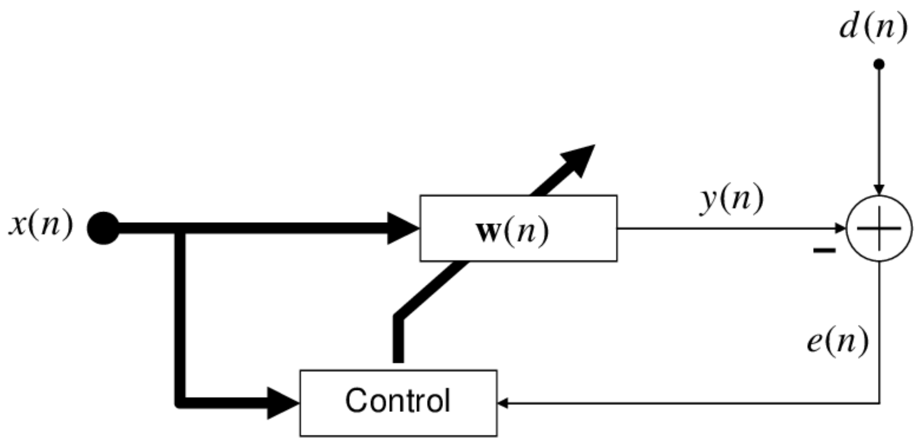

Figure 1: Block diagram of Adaptive Weiner Filter

In contrast, the RLS algorithm offers significantly faster convergence rates by leveraging all past input data. Introduced by Gauss in the 18th century and later adapted for signal processing, the RLS algorithm recursively minimizes a weighted least squares cost function, providing optimal filter coefficients at each step. This performance comes at the cost of increased computational complexity and memory requirements, making it less suitable for systems with limited resources.

This project aims to implement both LMS and RLS adaptive Wiener filters and compare their convergence behaviors. By examining their performance in various signal environments, we seek to highlight the strengths and limitations of each algorithm, providing insights into their suitability for different real-world applications. Through rigorous experimentation and analysis, this report will contribute to a deeper understanding of adaptive filtering techniques and their practical implications in modern signal processing.

2 Least Mean Squares - (LMS)

The LMS algorithm aims to minimize the mean squared error (MSE) between the desired signal d(n) and the estimated output y(n) by adjusting the filter coefficients iteratively.

Consider a linear adaptive filter with M taps:

$$y(n) = w^T(n)x(n)$$

where $x(n)$ is the input signal, $w(n) = [w_0(n), w_1(n), \ldots, w_{M-1}(n)]^T$ are the adaptive filter coefficients, and $y(n)$ is the output.

The error at time n is given by $e(n) = d(n) - y(n)$.

The LMS algorithm updates the weights using:

$$w(n+1) = w(n) + \mu e(n)x(n)$$

where μ is the step size or learning rate.

The block diagram of LMS algorithm based Adaptive is shown in Fig 1.

2.1 Implementation

Following is the verilog code for implementing LMS algorithm.

```

module LMS (
    input Clk, Rst,
    input signed [15:0] x_in, // Input signal
    output signed [15:0] y_out, // Output signal
    output signed [15:0] w0, w1, w2, w3, // Weights
    output signed [15:0] err // Errors
);
// Here 4 bits are used for decimal and 12 bits are used for representing fractions
// Declare gamma as a fixed-point value
  parameter signed [15:0] gamma = 16'b0000001100110011; // gamma = 0.2

reg signed [15:0] wn[0:3], x[0:3] ;
wire signed [15:0] wn_u[0:3];
wire signed [15:0] fir_out, m1,m2,m3,m4, m12,m22,m32,m42;
wire signed [31:0] m11,m21,m31,m41, m13,m23,m33,m43, y_out1;
reg signed [15:0] d_in;

FIR_Filter fir(
    .Clk(Clk),
    .Rst(Rst),
    .x(x_in),
    .d(fir_out)
  );

  always@(*)
    begin
      if(Rst)
```

```
        d_in <= 0;
      else
        d_in <= fir_out;
    end

always@(posedge Clk or posedge Rst)
    if(Rst)
    begin
        x[3]<=16'b0;
        x[2]<=16'b0;
        x[1]<=16'b0;
        x[0]<=16'b0;
    end
    else
    begin
        x[3]<=x[2];
        x[2]<=x[1];
        x[1]<=x[0];
        x[0]<=x_in;
    end
    assign y_out1 = $signed(wn[0])*$signed(x[0]) + $signed(wn[1])*$signed(x[1])
        + $signed(wn[2])*$signed(x[2]) + $signed(wn[3])*$signed(x[3]);
    assign y_out = y_out1[27:12];

assign m11 = $signed(err)*$signed(x[0]);
assign m12 = m11[27:12];
assign m13 = $signed(gamma)*$signed(m12);
assign m1 = m13[27:12];

assign m21 = $signed(err)*$signed(x[1]);
assign m22 = m21[27:12];
assign m23 = $signed(gamma)*$signed(m22);
assign m2 = m23[27:12];

assign m31 = $signed(err)*$signed(x[2]);
assign m32 = m31[27:12];
assign m33 = $signed(gamma)*$signed(m32);
assign m3 = m33[27:12];

assign m41 = $signed(err)*$signed(x[3]);
assign m42 = m41[27:12];
assign m43 = $signed(gamma)*$signed(m42);
assign m4 = m43[27:12];

assign wn_u[0] = wn[0] + m1;
assign wn_u[1] = wn[1] + m2;
assign wn_u[2] = wn[2] + m3;
assign wn_u[3] = wn[3] + m4;


always@( posedge Clk or posedge Rst)
    begin
    if(Rst)
        begin

         wn[0] <= 16'b0000000000000000;
         wn[1] <= 16'b0000000000000000;
         wn[2] <= 16'b0000000000000000;
         wn[3] <= 16'b0000000000000000;

        end
     else
        begin
```

```
          wn[0] <= wn_u[0];
          wn[1] <= wn_u[1];
          wn[2] <= wn_u[2];
          wn[3] <= wn_u[3];

        end
    end

    assign w0 = wn[0];
    assign w1 = wn[1];
    assign w2 = wn[2];
    assign w3 = wn[3];

    assign err = d_in - y_out;

endmodule



module FIR_Filter (
  input Clk,
  input Rst,
  input signed [15:0] x,
  output signed [15:0] d
);

  reg signed [15:0] w0, w1, w2, w3;
  reg signed [15:0] xn_0, xn_1, xn_2, xn_3;
  reg signed [31:0] d1;


  always @(posedge Clk)
  begin
    if (Rst)
        begin
          xn_0 <= 0;
          xn_1 <= 0;
          xn_2 <= 0;
          xn_3 <= 0;
          d1 <= 0;
        end
    else
        begin
          xn_3 <= xn_2;
          xn_2 <= xn_1;
          xn_1 <= xn_0;
          xn_0 <= x;
          d1 <= w0*xn_0 + w1*xn_1 + w2*xn_2 + w3*xn_3;
        end
  end

  assign d = d1[27:12];

  initial
  begin
    w0 =  16'b0000100000000000;
    w1 =  16'b0000100000000000;
    w2 =  16'b0000100000000000;
    w3 =  16'b0000100000000000;
  end

endmodule
```

2.2 Results

Following is the test bench used for LMS algorithm

```

module LMS_TB;
  reg Clk, Rst;
  reg signed [15:0] x_in;
  reg signed [15:0] x_gen;
  wire signed [15:0] y_out;
  reg[15:0] input_data[0:100];
  wire signed [15:0] d, err, w0, w1, w2, w3;
  integer i;

  localparam SF = 2.0**-12.0;

  LMS dut(
    .Clk(Clk),
    .Rst(Rst),
    .x_in(x_in),
    .w0(w0),
    .w1(w1),
    .w2(w2),
    .w3(w3),
    .y_out(y_out),
    .err(err)
  );

  FIR_Filter F1(
    .Clk(Clk),
    .Rst(Rst),
    .x(x_in),
    .d(d)
  );

  always begin
    #5 Clk = ~Clk;
  end

   initial begin
    drive_reset();

     $readmemb("LMS_inputs.txt", input_data);

     for( i=0; i<100;i=i+1)
        begin
          drive_input(input_data[i]);
          check_output();
        end

      repeat(30)@(negedge Clk)
      $finish;
    end


    task drive_reset();
    $display ("Driving the reset");
    Clk <= 1'b0;
    x_in<= 0;
    @ (negedge Clk)
    Rst = 1;
    @ (posedge Clk)
    Rst = 1;

```

```
    @ (negedge Clk)
    Rst = 0;
  endtask

  task drive_input(input [15:0] x_gen);
    $display ("Recieved the ready signal and driving the input");
    @ (negedge Clk)
    x_in = x_gen;
    $display(x_in*SF);
    $display ($itor(x_in*SF));
  endtask

  task check_output();
    $display("Time: %f, Iteration: %f, Input: %f, Output: %f, Error: %f w0: %f,
         w1 : %f, w2 : %f, w3 : %f", $time, i, x_in*SF, y_out*SF, err*SF,w0*SF,
       w1*SF,w2*SF,w3*SF);

  endtask

  initial begin
  $dumpfile("dump_lms.vcd");
  $dumpvars;
  end

endmodule
```

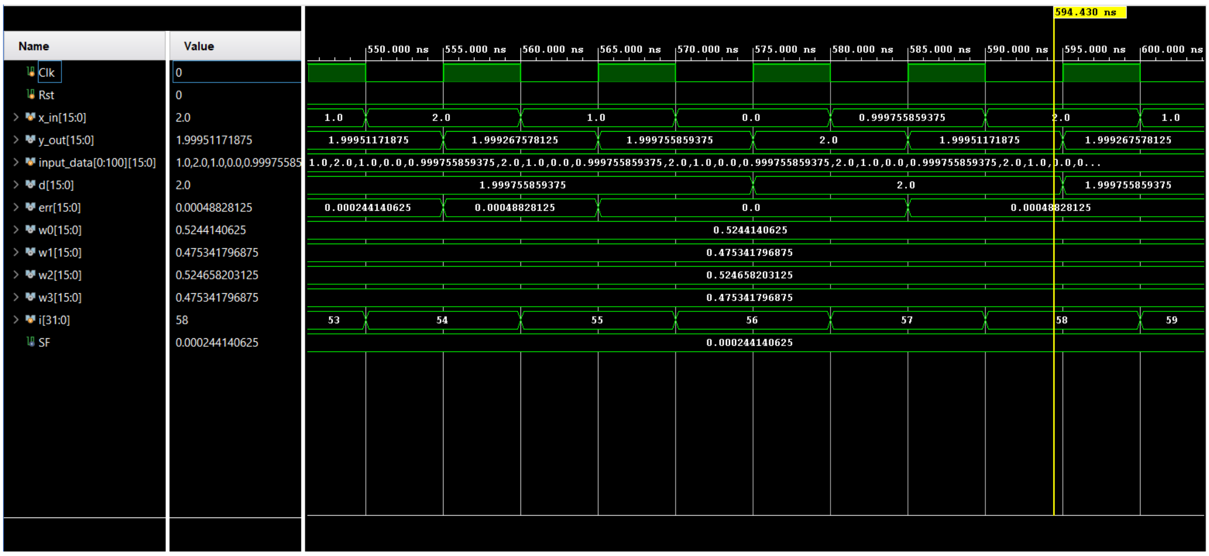

Figure 2: Timing Diagram of LMS algorithm

```
Time: 750.000000, Iteration: 73.000000, Input: 2.000000, Output: 1.999512, Error: 0.000488 w0: 0.524414, w1 : 0.475342, w2 : 0.524658, w3 : 0.475342
Recieved the ready signal and driving the input
1
1
Time: 760.000000, Iteration: 74.000000, Input: 1.000000, Output: 1.999268, Error: 0.000488 w0: 0.524414, w1 : 0.475342, w2 : 0.524658, w3 : 0.475342
Recieved the ready signal and driving the input
0
0
Time: 770.000000, Iteration: 75.000000, Input: 0.000000, Output: 1.999512, Error: 0.000244 w0: 0.524414, w1 : 0.475342, w2 : 0.524658, w3 : 0.475342
Recieved the ready signal and driving the input
0.999756
0.999756
Time: 780.000000, Iteration: 76.000000, Input: 0.999756, Output: 1.999756, Error: 0.000000 w0: 0.524414, w1 : 0.475342, w2 : 0.524658, w3 : 0.475342
Recieved the ready signal and driving the input
2
2
Time: 790.000000, Iteration: 77.000000, Input: 2.000000, Output: 1.999512, Error: 0.000244 w0: 0.524414, w1 : 0.475342, w2 : 0.524658, w3 : 0.475342
Recieved the ready signal and driving the input
1
1
Time: 800.000000, Iteration: 78.000000, Input: 1.000000, Output: 1.999268, Error: 0.000488 w0: 0.524414, w1 : 0.475342, w2 : 0.524658, w3 : 0.475342
Recieved the ready signal and driving the input
```

Figure 3: Displaying the values in each iteration of LMS algorithm

Recursive Least Squares (RLS) Algorithm

The Recursive Least Squares (RLS) algorithm operates on the principle of recursively updating the filter coefficients to minimize the weighted least squares error. At each time instant n, RLS computes the filter coefficients by recursively updating an estimate of the inverse correlation matrix and the filter weights based on the current input, output, and desired signal.

The key idea behind RLS is to maintain an estimate of the inverse correlation matrix $\mathbf{P}(n)$, which captures the statistical properties of the input signals. By updating this estimate recursively using the current input and the forgetting factor λ, RLS adapts to changing system dynamics and optimally tracks the input statistics over time.

Additionally, RLS computes the gain vector $\mathbf{k}(n)$ to adjust the filter weights based on the current estimation error. This gain vector ensures that the filter adapts quickly to changes in the input-output relationship while maintaining stability and robustness against noise and disturbances.

Overall, the recursive nature of RLS enables it to efficiently track time-varying systems and achieve fast convergence to the optimal filter coefficients, making it a powerful tool in adaptive signal processing applications.

Algorithmn

Initialization

Initialize the algorithm by setting:

$$\mathbf{w}(0) = \mathbf{0}$$

$$\mathbf{P}(0) = \delta^{-1}\mathbf{I}$$

Where δ is a constant

- Use a large positive constant for low SNR (Signal-to-Noise Ratio).
- Use a small positive constant for high SNR.

Now for n= 0,1,2... we'll do the following steps

Filter Output

The filter output $y(n)$ is computed as:

$$y(n) = \mathbf{w}^H(n-1)\mathbf{u}(n)$$

Error Signal

The error signal $e(n)$ is calculated by subtracting the filter output $y(n)$ from the desired signal $d(n)$:

$$e(n) = d(n) - y(n)$$

Gain Vector Update

The gain vector $\mathbf{k}(n)$ is updated using the previous inverse correlation matrix $\mathbf{P}(n-1)$ and the current input vector $\mathbf{u}(n)$:

$$\mathbf{k}(n) = \frac{\mathbf{P}(n-1)\mathbf{u}(n)}{\lambda + \mathbf{u}^H(n)\mathbf{P}(n-1)\mathbf{u}(n)}$$

Weight Update

The filter weights $\mathbf{w}(n)$ are updated using the previous weights $\mathbf{w}(n-1)$, the gain vector $\mathbf{k}(n)$, and the error signal $e(n)$:

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{k}(n)e^H(n)$$

Inverse Correlation Matrix Update

Finally, the inverse correlation matrix $\mathbf{P}(n)$ is updated as follows:

$$\mathbf{P}(n) = \lambda^{-1}\mathbf{P}(n-1) - \lambda^{-1}\mathbf{k}(n)\mathbf{u}^H(n)\mathbf{P}(n-1)$$

Implementation

```
module RLS (
    input Clk, Rst,
    input signed [15:0] x_in, // Input signal
    output signed [15:0] y_out, // Output signal
    output signed [15:0] w0, w1, w2, w3, // Weights
    output signed [15:0] err, // Errors
  output signed [15:0] p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14, p15, p16 // Auto correlation Matrix

);
// Here 4 bits are used for decimal and 12 bits are used for representing fractions

  parameter signed [15:0] gamma = 16'b0000111100111111, gamma_inverse = 16'b0001000011000101 ; // gamma = 0.95


  reg signed [15:0]  p[0:15]; // Inverse of Auto Correlarion Function
  wire signed [15:0]  p_u[0:15], r[0:15] , p_u_raw_wire[0:15]; // Temp value for storing past values of p
  reg signed [15:0] k[0:3]; // Gain Vector


  reg signed [15:0] wn[0:3], x[0:3] ;
  wire signed [15:0] wn_u[0:3], k_1[0:3], q[0:3];
  wire signed [15:0] m1,m2,m3,m4,l, k_scale;
  wire signed [31:0] m11, m21, m31, m41, y_out1, p_11, p_12, p_13, p_14, p_21, p_22, p_23, p_24, p_31, p_32, p_33, p_34, p_41, p_42, p_43, p_44;
  reg signed [15:0] d_in;

  FIR_Filter fir(
    .Clk(Clk),
```

```
    .Rst(Rst),
    .x(x_in),
    .d(d_in)
  );


  MAT_VECT_Mul M1(
    .Mat(p),
    .Vect(x),
    .Vect_out(k_1)
  );


  VECT_VECT_Mul M2(
    .Vect1(x),
    .Vect2(k_1),
    .Vect_out(l)
  );


  VECT_MAT_Mul M3(
    .Mat(p),
    .Vect(x),
    .Vect_out(q)
  );


  VECT_VECT_2Mul M4(
    .Vect1(k),
    .Vect2(q),
    .Mat_out(r)
  );


   Divider D0 (
     .dividend(k_1[0]),
     .divisor(k_scale),
     .quotient(k[0])
   );

   Divider D1 (
     .dividend(k_1[1]),
     .divisor(k_scale),
     .quotient(k[1])
   );

   Divider D2 (
     .dividend(k_1[2]),
     .divisor(k_scale),
     .quotient(k[2])
   );

   Divider D3 (
     .dividend(k_1[3]),
     .divisor(k_scale),
     .quotient(k[3])
   );



  always@(posedge Clk or posedge Rst)
      if(Rst)
      begin // Initializing the input x[-1] as 16'b0
```

```

            x[3]<=16'b0;
            x[2]<=16'b0;
            x[1]<=16'b0;
            x[0]<=16'b0;

        end
        else
        begin // Updating the inputs

            x[3]<=x[2];
            x[2]<=x[1];
            x[1]<=x[0];
            x[0]<=x_in;

        end



always@( posedge Clk or posedge Rst)
    begin
    if(Rst)
        begin // Initializing the guess for the weight as wn[-1] as 16'b0

            wn[0]  <=  16'b0;
            wn[1]  <=  16'b0;
            wn[2]  <=  16'b0;
            wn[3]  <=  16'b0;

        end
     else
        begin   // Updating the weights

          wn[0]  <=  wn_u[0];
          wn[1]  <=  wn_u[1];
          wn[2]  <=  wn_u[2];
          wn[3]  <=  wn_u[3];

        end
    end




always@( posedge Clk or posedge Rst)
    begin
    if(Rst)
        begin // Initializing the value for the inverse of autocorrelation as p[-1] = (1/d)I as d tends to zero

            p[0]  <=  16'b0000000011111111;
            p[1]  <=  16'b0000000000000000;
            p[2]  <=  16'b0000000000000000;
            p[3]  <=  16'b0000000000000000;
            p[4]  <=  16'b0000000000000000;
            p[5]  <=  16'b0000000011111111;
            p[6]  <=  16'b0000000000000000;
            p[7]  <=  16'b0000000000000000;
            p[8]  <=  16'b0000000000000000;
            p[9]  <=  16'b0000000000000000;
            p[10]  <=  16'b0000000011111111;
            p[11]  <=  16'b0000000000000000;
            p[12]  <=  16'b0000000000000000;
```

```
            p[13] <= 16'b0000000000000000;
            p[14] <= 16'b0000000000000000;
            p[15] <= 16'b0000000011111111;

        end
     else
        begin    // Updating the inverse of autocorrelation

            p[0] <= p_u[0];
            p[1] <= p_u[1];
            p[2] <= p_u[2];
            p[3] <= p_u[3];
            p[4] <= p_u[4];
            p[5] <= p_u[5];
            p[6] <= p_u[6];
            p[7] <= p_u[7];
            p[8] <= p_u[8];
            p[9] <= p_u[9];
            p[10] <= p_u[10];
            p[11] <= p_u[11];
            p[12] <= p_u[12];
            p[13] <= p_u[13];
            p[14] <= p_u[14];
            p[15] <= p_u[15];

        end
    end

    assign y_out1 = $signed(wn[0])*$signed(x[0]) + $signed(wn[1])*$signed(x[1])
         + $signed(wn[2])*$signed(x[2]) + $signed(wn[3])*$signed(x[3]);
    assign y_out = y_out1[27:12];

    assign err = d_in - y_out;

    assign k_scale = gamma + l;

    assign m11 = $signed(err)*$signed(k[0]);
    assign m1 = m11[27:12];

    assign m21 = $signed(err)*$signed(k[1]);
    assign m2 = m21[27:12];

    assign m31 = $signed(err)*$signed(k[2]);
    assign m3 = m31[27:12];

    assign m41 = $signed(err)*$signed(k[3]);
    assign m4 = m41[27:12];


    assign wn_u[0] = wn[0] + m1;
    assign wn_u[1] = wn[1] + m2;
    assign wn_u[2] = wn[2] + m3;
    assign wn_u[3] = wn[3] + m4;


    assign w0 = wn[0];
    assign w1 = wn[1];
    assign w2 = wn[2];
    assign w3 = wn[3];

    assign p1 = p_u[0];
    assign p2 = p_u[1];
    assign p3 = p_u[2];
```

```
    assign p4 = p_u[3];
    assign p5 = p_u[4];
    assign p6 = p_u[5];
    assign p7 = p_u[6];
    assign p8 = p_u[7];
    assign p9 = p_u[8];
    assign p10 = p_u[9];
    assign p11 = p_u[10];
    assign p12 = p_u[11];
    assign p13 = p_u[12];
    assign p14 = p_u[13];
    assign p15 = p_u[14];
    assign p16 = p_u[15];

    assign p_11 = gamma_inverse * p_u_raw_wire[0];
    assign p_u[0] = p_11[27:12];

    assign p_12 = gamma_inverse * p_u_raw_wire[1];
    assign p_u[1] = p_12[27:12];

    assign p_13 = gamma_inverse * p_u_raw_wire[2];
    assign p_u[2] = p_13[27:12];

    assign p_14 = gamma_inverse * p_u_raw_wire[3];
    assign p_u[3] = p_14[27:12];

    assign p_21 = gamma_inverse * p_u_raw_wire[4];
    assign p_u[4] = p_21[27:12];

    assign p_22 = gamma_inverse * p_u_raw_wire[5];
    assign p_u[5] = p_22[27:12];

    assign p_23 = gamma_inverse * p_u_raw_wire[6];
    assign p_u[6] = p_23[27:12];

    assign p_24 = gamma_inverse * p_u_raw_wire[7];
    assign p_u[7] = p_24[27:12];

    assign p_31 = gamma_inverse * p_u_raw_wire[8];
    assign p_u[8] = p_31[27:12];

    assign p_32 = gamma_inverse * p_u_raw_wire[9];
    assign p_u[9] = p_32[27:12];

    assign p_33 = gamma_inverse * p_u_raw_wire[10];
    assign p_u[10] = p_33[27:12];

    assign p_34 = gamma_inverse * p_u_raw_wire[11];
    assign p_u[11] = p_34[27:12];

    assign p_41 = gamma_inverse * p_u_raw_wire[12];
    assign p_u[12] = p_41[27:12];

    assign p_42 = gamma_inverse * p_u_raw_wire[13];
    assign p_u[13] = p_42[27:12];

    assign p_43 = gamma_inverse * p_u_raw_wire[14];
    assign p_u[14] = p_43[27:12];

    assign p_44 = gamma_inverse * p_u_raw_wire[15];
    assign p_u[15] = p_44[27:12];
```

```
    assign p_u_raw_wire[0] = p[0] - r[0];
    assign p_u_raw_wire[1] = p[1] - r[1];
    assign p_u_raw_wire[2] = p[2] - r[2];
    assign p_u_raw_wire[3] = p[3] - r[3];
    assign p_u_raw_wire[4] = p[4] - r[4];
    assign p_u_raw_wire[5] = p[5] - r[5];
    assign p_u_raw_wire[6] = p[6] - r[6];
    assign p_u_raw_wire[7] = p[7] - r[7];
    assign p_u_raw_wire[8] = p[8] - r[8];
    assign p_u_raw_wire[9] = p[9] - r[9];
    assign p_u_raw_wire[10] = p[10] - r[10];
    assign p_u_raw_wire[11] = p[11] - r[11];
    assign p_u_raw_wire[12] = p[12] - r[12];
    assign p_u_raw_wire[13] = p[13] - r[13];
    assign p_u_raw_wire[14] = p[14] - r[14];
    assign p_u_raw_wire[15] = p[15] - r[15];



endmodule






// This is a Multiplier for a 4*4 Matrix to a 4*1 Vector
module MAT_VECT_Mul(
  input signed [15:0] Mat[0:15],
  input signed [15:0] Vect[0:3],
  output signed [15:0] Vect_out[0:3]
);
  wire signed [31:0] Vect_out1, Vect_out2, Vect_out3, Vect_out4;

  assign Vect_out1 = $signed(Vect[0])*$signed(Mat[0]) + $signed(Vect[1])*
      $signed(Mat[1]) + $signed(Vect[2])*$signed(Mat[2]) + $signed(Vect[3])*
      $signed(Mat[3]);
  assign Vect_out[0] = Vect_out1[27:12];
  assign Vect_out2 = $signed(Vect[0])*$signed(Mat[4]) + $signed(Vect[1])*
      $signed(Mat[5]) + $signed(Vect[2])*$signed(Mat[6]) + $signed(Vect[3])*
      $signed(Mat[7]);
  assign Vect_out[1] = Vect_out2[27:12];
  assign Vect_out3 = $signed(Vect[0])*$signed(Mat[8]) + $signed(Vect[1])*
      $signed(Mat[9]) + $signed(Vect[2])*$signed(Mat[10]) + $signed(Vect[3])*
      $signed(Mat[11]);
  assign Vect_out[2] = Vect_out3[27:12];
  assign Vect_out4 = $signed(Vect[0])*$signed(Mat[12]) + $signed(Vect[1])*
      $signed(Mat[13]) + $signed(Vect[2])*$signed(Mat[14]) + $signed(Vect[3])*
      $signed(Mat[15]);
  assign Vect_out[3] = Vect_out4[27:12];

endmodule






// This is a Multiplier for a 1*4 Vector to a 4*4 Matrix
module VECT_MAT_Mul(
  input signed [15:0] Mat[0:15],
  input signed [15:0] Vect[0:3],
```

```
  output signed [15:0] Vect_out[0:3]
);
  wire signed [31:0] Vect_out1, Vect_out2, Vect_out3, Vect_out4;

  assign Vect_out1 = $signed(Vect[0])*$signed(Mat[0]) + $signed(Vect[1])*$signed(Mat[4]) +  $signed(Vect[2])*$signed(Mat[8]) +  $signed(Vect[3])*$signed(Mat[12]);
  assign Vect_out[0] = Vect_out1[27:12];
  assign Vect_out2 = $signed(Vect[0])*$signed(Mat[1]) + $signed(Vect[1])*$signed(Mat[5]) +  $signed(Vect[2])*$signed(Mat[9]) +  $signed(Vect[3])*$signed(Mat[13]);
  assign Vect_out[1] = Vect_out2[27:12];
  assign Vect_out3 = $signed(Vect[0])*$signed(Mat[2]) + $signed(Vect[1])*$signed(Mat[6]) +  $signed(Vect[2])*$signed(Mat[10]) +  $signed(Vect[3])*$signed(Mat[14]);
  assign Vect_out[2] = Vect_out3[27:12];
  assign Vect_out4 = $signed(Vect[0])*$signed(Mat[3]) + $signed(Vect[1])*$signed(Mat[7]) +  $signed(Vect[2])*$signed(Mat[11]) +  $signed(Vect[3])*$signed(Mat[15]);
  assign Vect_out[3] = Vect_out4[27:12];

endmodule








// This is a Multiplier for a 1*4 Vector to a 4*1 Vector
module VECT_VECT_Mul(
  input signed [15:0] Vect1[0:3],
  input signed [15:0] Vect2[0:3],
  output signed [15:0] Vect_out
);
  wire signed [31:0] Vect_out1;

  assign Vect_out1 = $signed(Vect1[0])*$signed(Vect2[0]) + $signed(Vect1[1])*$signed(Vect2[1]) +  $signed(Vect1[2])*$signed(Vect2[2]) +  $signed(Vect1[3])*$signed(Vect2[3]);
  assign Vect_out = Vect_out1[27:12];

endmodule







// This is a Multiplier for a 4*1 Vector to a 1*4 Vector
module VECT_VECT_2Mul(
  input signed [15:0] Vect1[0:3],
  input signed [15:0] Vect2[0:3],
  output signed [15:0] Mat_out[0:15]
);
  wire signed [31:0] Mat_out1[0:15];

  assign Mat_out1[0] = $signed(Vect1[0])*$signed(Vect2[0]);
  assign Mat_out[0] = Mat_out1[0][27:12];
  assign Mat_out1[1] = $signed(Vect1[0])*$signed(Vect2[1]);
  assign Mat_out[1] = Mat_out1[1][27:12];
```

```
  assign Mat_out1[2] = $signed(Vect1[0])*$signed(Vect2[2]);
  assign Mat_out[2] = Mat_out1[2][27:12];
  assign Mat_out1[3] = $signed(Vect1[0])*$signed(Vect2[3]);
  assign Mat_out[3] = Mat_out1[3][27:12];
  assign Mat_out1[4] = $signed(Vect1[1])*$signed(Vect2[0]);
  assign Mat_out[4] = Mat_out1[4][27:12];
  assign Mat_out1[5] = $signed(Vect1[1])*$signed(Vect2[1]);
  assign Mat_out[5] = Mat_out1[5][27:12];
  assign Mat_out1[6] = $signed(Vect1[1])*$signed(Vect2[2]);
  assign Mat_out[6] = Mat_out1[6][27:12];
  assign Mat_out1[7] = $signed(Vect1[1])*$signed(Vect2[3]);
  assign Mat_out[7] = Mat_out1[7][27:12];
  assign Mat_out1[8] = $signed(Vect1[2])*$signed(Vect2[0]);
  assign Mat_out[8] = Mat_out1[8][27:12];
  assign Mat_out1[9] = $signed(Vect1[2])*$signed(Vect2[1]);
  assign Mat_out[9] = Mat_out1[9][27:12];
  assign Mat_out1[10] = $signed(Vect1[2])*$signed(Vect2[2]);
  assign Mat_out[10] = Mat_out1[10][27:12];
  assign Mat_out1[11] = $signed(Vect1[2])*$signed(Vect2[3]);
  assign Mat_out[11] = Mat_out1[11][27:12];
  assign Mat_out1[12] = $signed(Vect1[3])*$signed(Vect2[0]);
  assign Mat_out[12] = Mat_out1[12][27:12];
  assign Mat_out1[13] = $signed(Vect1[3])*$signed(Vect2[1]);
  assign Mat_out[13] = Mat_out1[13][27:12];
  assign Mat_out1[14] = $signed(Vect1[3])*$signed(Vect2[2]);
  assign Mat_out[14] = Mat_out1[14][27:12];
  assign Mat_out1[15] = $signed(Vect1[3])*$signed(Vect2[3]);
  assign Mat_out[15] = Mat_out1[15][27:12];

endmodule





// Divider for Signed bit fixed point numbers

module Divider(
    input signed [15:0] dividend,
    input signed [15:0] divisor,
    output signed [15:0] quotient
);
    reg signed [31:0] scaled_dividend;
    reg signed [15:0] result;

    always @(*) begin
        if (divisor == 0) begin
            result = 16'b0111111111111111; // Max positive value to indicate error
        end else begin
            scaled_dividend = dividend <<< 12;
            result = scaled_dividend / divisor;
        end
    end

    assign quotient = result;
endmodule





// FIR Filter Module
```

```
module FIR_Filter (
  input Clk,
  input Rst,
  input signed [15:0] x,
  output signed [15:0] d
);

  reg signed [15:0] w0, w1, w2, w3;
  reg signed [15:0] xn_0, xn_1, xn_2, xn_3;
  reg signed [31:0] d1;


  always @(posedge Clk)
  begin
    if (Rst)
        begin
          xn_0 <= 0;
          xn_1 <= 0;
          xn_2 <= 0;
          xn_3 <= 0;
          d1 <= 0;
        end
    else
        begin
          xn_3 <= xn_2;
          xn_2 <= xn_1;
          xn_1 <= xn_0;
          xn_0 <= x;
          d1 <= w0*xn_0 + w1*xn_1 + w2*xn_2 + w3*xn_3;
        end
  end

  assign d = d1[27:12];

  initial
  begin
    w0 =  16'b0000100000000000;
    w1 =  16'b0000100000000000;
    w2 =  16'b0000100000000000;
    w3 =  16'b0000100000000000;
  end

endmodule
```

Results

Following is the test bench used for LMS algorithm

```
module RLS_TB;
  reg Clk, Rst;
  reg signed [15:0] x_in;
  reg signed [15:0] x_gen;
  wire signed [15:0] y_out;
  reg[15:0] input_data[0:100];
  wire signed [15:0] d, err, w0, w1, w2, w3;
  integer o;
  wire signed [15:0] p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13,
      p14, p15, p16;

  localparam SF = 2.0**-12.0;

  RLS dut(
    .Clk(Clk),
```

```
    .Rst(Rst),
    .x_in(x_in),
    .w0(w0),
    .w1(w1),
    .w2(w2),
    .w3(w3),
    .y_out(y_out),
    .err(err),
    .p1(p1),
    .p2(p2),
    .p3(p3),
    .p4(p4),
    .p5(p5),
    .p6(p6),
    .p7(p7),
    .p8(p8),
    .p9(p9),
    .p10(p10),
    .p11(p11),
    .p12(p12),
    .p13(p13),
    .p14(p14),
    .p15(p15),
    .p16(p16)
  );

  FIR_Filter F1(
    .Clk(Clk),
    .Rst(Rst),
    .x(x_in),
    .d(d)
  );

  always begin
    #10 Clk = ~Clk;
  end

   initial begin
    drive_reset();

     $readmemb("RLS_inputs.txt", input_data);

     for( o=0; o<100;o=o+1)
        begin
          drive_input(input_data[o]);
          check_output();
        end

      repeat(30)@(negedge Clk)
      $finish;
    end


    task drive_reset();
    $display ("Driving the reset");
    Clk <= 1'b0;
    x_in<= 0;
    @ (negedge Clk)
    Rst = 1;
    @ (posedge Clk)
    Rst = 1;
    @ (negedge Clk)
    Rst = 0;
```

```verilog
  endtask

  task drive_input(input [15:0] x_gen);
    $display ("Recieved the ready signal and driving the input");
    @ (negedge Clk)
    x_in = x_gen;
    $display(x_in*SF);
    $display ($itor(x_in*SF));
  endtask

  task check_output();
    $display("Time: %f, Iteration: %f, Input: %f, Output: %f, Error: %f, w0 : %
        f, w1 : %f, w2 : %f, w3 : %f", $time, o, x_in*SF, y_out*SF, err*SF, w0*
        SF, w1*SF, w2*SF, w3*SF);

  endtask

  initial begin
  $dumpfile("dump_rls.vcd");
  $dumpvars;
  end

endmodule
```

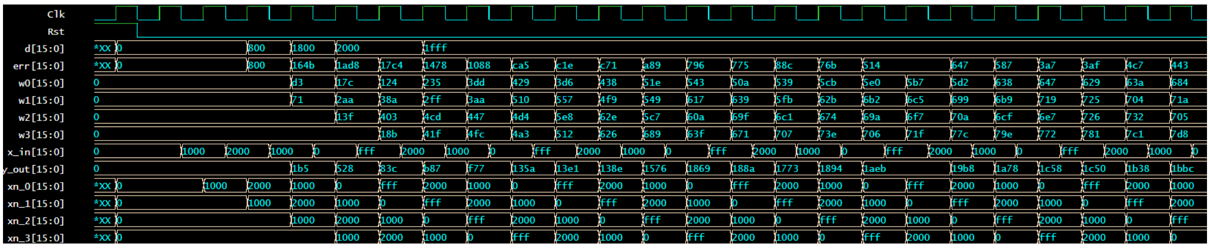

Figure 4: Timing Diagram of RLS algorithm

```
# KERNEL: 2
# KERNEL: Time: 860.000000, Iteration: 41.000000, Input: 2.000000, Output: 1.932129, Error: 0.067627, w0 : 0.442871, w1 : 0.500244, w2 : 0.465088, w3 : 0.512207
# KERNEL: Recieved the ready signal and driving the input
# KERNEL: 0.999755859375
# KERNEL: 0.999755859375
# KERNEL: Time: 880.000000, Iteration: 42.000000, Input: 0.999756, Output: 1.901123, Error: 0.098633, w0 : 0.443604, w1 : 0.498047, w2 : 0.465576, w3 : 0.516113
# KERNEL: Recieved the ready signal and driving the input
# KERNEL: 0
# KERNEL: 0
# KERNEL: Time: 900.000000, Iteration: 43.000000, Input: 0.000000, Output: 1.909668, Error: 0.090088, w0 : 0.448975, w1 : 0.499512, w2 : 0.461914, w3 : 0.517578
# KERNEL: Recieved the ready signal and driving the input
# KERNEL: 0.999755859375
# KERNEL: 0.999755859375
# KERNEL: Time: 920.000000, Iteration: 44.000000, Input: 0.999756, Output: 1.944824, Error: 0.054932, w0 : 0.449707, w1 : 0.504883, w2 : 0.462891, w3 : 0.514648
# KERNEL: Recieved the ready signal and driving the input
# KERNEL: 2
# KERNEL: 2
# KERNEL: Time: 940.000000, Iteration: 45.000000, Input: 2.000000, Output: 1.944092, Error: 0.055664, w0 : 0.447998, w1 : 0.505371, w2 : 0.466064, w3 : 0.515137
# KERNEL: Recieved the ready signal and driving the input
# KERNEL: 1
# KERNEL: 1
# KERNEL: Time: 960.000000, Iteration: 46.000000, Input: 1.000000, Output: 1.918701, Error: 0.081055, w0 : 0.448730, w1 : 0.503418, w2 : 0.466553, w3 : 0.518311
# KERNEL: Recieved the ready signal and driving the input
# KERNEL: 0
# KERNEL: 0
# KERNEL: Time: 980.000000, Iteration: 47.000000, Input: 0.000000, Output: 1.925781, Error: 0.073975, w0 : 0.453125, w1 : 0.504639, w2 : 0.463623, w3 : 0.519287
# KERNEL: Recieved the ready signal and driving the input
# KERNEL: 0.999755859375
# KERNEL: 0.999755859375
# KERNEL: Time: 1000.000000, Iteration: 48.000000, Input: 0.999756, Output: 1.954346, Error: 0.045410, w0 : 0.453613, w1 : 0.509033, w2 : 0.464355, w3 : 0.516846
```

Figure 5: Displaying the values in each iteration of RLS algorithm

3 Conclusions

In this project we have implemented LMS and RLS algorithms. The LMS algorithm demonstrated robustness in scenarios with less computational requirements. Its performance is heavily dependent on the choice of step size (μ), which affects both the convergence rate and stability. The RLS algorithm is more complex and computationally intensive and showcased better performance in terms of convergence speed. It adapts more quickly to changes in the signal environment due to its recursive formulation and consideration of past errors.

From the results we see that LMS algorithm took around 50 iterations to converge to optimum weights whereas RLS took only 27 iterations to converge. This shows that RLS has better performance than LMS but at the tradeoff of computational complexity.

LMS Algorithm	**RLS Algorithm**
Simple and easy to apply.	More complex and computationally expensive.
Slower convergence.	Faster convergence.
Adapts using gradient-based approach for filter weight updates.	Adapts using recursive approach to minimize weighted least squares cost.
Doesn't account for past data.	Accounts for all past data, using a forgetting factor to de-emphasize older data.
Objective: minimize current mean square error.	Objective: minimize total weighted squared error.
No memory involved; older errors don't affect total error.	Infinite memory; all errors considered with option to de-emphasize older data.

Table 1: Comparison of LMS and RLS Algorithms