

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.

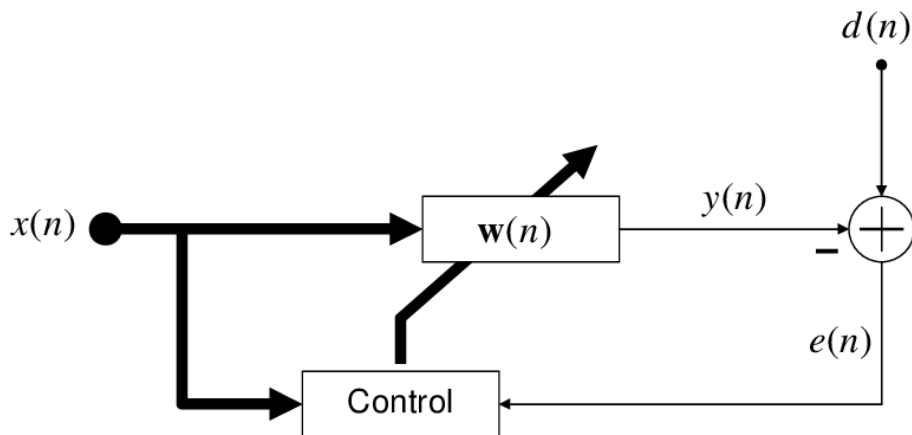


Figure 1: Block diagram of Adaptive Wiener Filter

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 algo-

rithm 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, $w(n) = [w_0(n), w_1(n), \dots, w_{M-1}(n)]^T$ 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.

```

1
2 module LMS (
3     input Clk, Rst,
4     input signed [15:0] x_in, // Input signal
5     output signed [15:0] y_out, // Output signal
6     output signed [15:0] w0, w1, w2, w3, // Weights
7     output signed [15:0] err // Errors
8 );
9 // Here 4 bits are used for decimal and 12 bits are used for representing
   fractions
10 // Declare gamma as a fixed-point value
11 parameter signed [15:0] gamma = 16'b00000001100110011; // gamma = 0.2
12
13 reg signed [15:0] wn[0:3], x[0:3] ;
14 wire signed [15:0] wn_u[0:3];
15 wire signed [15:0] fir_out, m1,m2,m3,m4, m12,m22,m32,m42;
16 wire signed [31:0] m11,m21,m31,m41, m13,m23,m33,m43, y_out1;
17 reg signed [15:0] d_in;
18
19 FIR_Filter fir(
20     .Clk(Clk),
21     .Rst(Rst),
22     .x(x_in),
23     .d(fir_out)
24 );
25
26 always@(*)
27 begin
28     if(Rst)

```

```

29         d_in <= 0;
30     else
31         d_in <= fir_out;
32     end
33
34 always@(posedge Clk or posedge Rst)
35     if(Rst)
36     begin
37         x[3] <= 16'b0;
38         x[2] <= 16'b0;
39         x[1] <= 16'b0;
40         x[0] <= 16'b0;
41     end
42     else
43     begin
44         x[3] <= x[2];
45         x[2] <= x[1];
46         x[1] <= x[0];
47         x[0] <= x_in;
48     end
49     assign y_out1 = $signed(wn[0])*$signed(x[0]) + $signed(wn[1])*$signed(x[1])
50         + $signed(wn[2])*$signed(x[2]) + $signed(wn[3])*$signed(x[3]);
51     assign y_out = y_out1[27:12];
52
53 assign m11 = $signed(err)*$signed(x[0]);
54 assign m12 = m11[27:12];
55 assign m13 = $signed(gamma)*$signed(m12);
56 assign m1 = m13[27:12];
57
58 assign m21 = $signed(err)*$signed(x[1]);
59 assign m22 = m21[27:12];
60 assign m23 = $signed(gamma)*$signed(m22);
61 assign m2 = m23[27:12];
62
63 assign m31 = $signed(err)*$signed(x[2]);
64 assign m32 = m31[27:12];
65 assign m33 = $signed(gamma)*$signed(m32);
66 assign m3 = m33[27:12];
67
68 assign m41 = $signed(err)*$signed(x[3]);
69 assign m42 = m41[27:12];
70 assign m43 = $signed(gamma)*$signed(m42);
71 assign m4 = m43[27:12];
72
73 assign wn_u[0] = wn[0] + m1;
74 assign wn_u[1] = wn[1] + m2;
75 assign wn_u[2] = wn[2] + m3;
76 assign wn_u[3] = wn[3] + m4;
77
78 always@(posedge Clk or posedge Rst)
79     begin
80     if(Rst)
81     begin
82
83         wn[0] <= 16'b0000000000000000;
84         wn[1] <= 16'b0000000000000000;
85         wn[2] <= 16'b0000000000000000;
86         wn[3] <= 16'b0000000000000000;
87
88     end
89     else
90     begin

```

```

91         wn[0] <= wn_u[0];
92         wn[1] <= wn_u[1];
93         wn[2] <= wn_u[2];
94         wn[3] <= wn_u[3];
95
96     end
97 end
98
99
100 assign w0 = wn[0];
101 assign w1 = wn[1];
102 assign w2 = wn[2];
103 assign w3 = wn[3];
104
105 assign err = d_in - y_out;
106
107 endmodule
108
109
110
111 module FIR_Filter (
112     input Clk,
113     input Rst,
114     input signed [15:0] x,
115     output signed [15:0] d
116 );
117
118 reg signed [15:0] w0, w1, w2, w3;
119 reg signed [15:0] xn_0, xn_1, xn_2, xn_3;
120 reg signed [31:0] d1;
121
122
123 always @(posedge Clk)
124 begin
125     if (Rst)
126     begin
127         xn_0 <= 0;
128         xn_1 <= 0;
129         xn_2 <= 0;
130         xn_3 <= 0;
131         d1 <= 0;
132     end
133     else
134     begin
135         xn_3 <= xn_2;
136         xn_2 <= xn_1;
137         xn_1 <= xn_0;
138         xn_0 <= x;
139         d1 <= w0*xn_0 + w1*xn_1 + w2*xn_2 + w3*xn_3;
140     end
141 end
142
143 assign d = d1[27:12];
144
145 initial
146 begin
147     w0 = 16'b0000100000000000;
148     w1 = 16'b0000100000000000;
149     w2 = 16'b0000100000000000;
150     w3 = 16'b0000100000000000;
151 end
152
153 endmodule

```

2.2 Results

Following is the test bench used for LMS algorithm

```
1
2 module LMS_TB;
3     reg Clk, Rst;
4     reg signed [15:0] x_in;
5     reg signed [15:0] x_gen;
6     wire signed [15:0] y_out;
7     reg [15:0] input_data[0:100];
8     wire signed [15:0] d, err, w0, w1, w2, w3;
9     integer i;
10
11     localparam SF = 2.0**-12.0;
12
13     LMS dut(
14         .Clk(Clk),
15         .Rst(Rst),
16         .x_in(x_in),
17         .w0(w0),
18         .w1(w1),
19         .w2(w2),
20         .w3(w3),
21         .y_out(y_out),
22         .err(err)
23     );
24
25     FIR_Filter F1(
26         .Clk(Clk),
27         .Rst(Rst),
28         .x(x_in),
29         .d(d)
30     );
31
32     always begin
33         #5 Clk = ~Clk;
34     end
35
36     initial begin
37         drive_reset();
38
39         $readmemb("LMS_inputs.txt", input_data);
40
41         for( i=0; i<100;i=i+1)
42             begin
43                 drive_input(input_data[i]);
44                 check_output();
45             end
46
47         repeat(30)@(negedge Clk)
48             $finish;
49     end
50
51
52     task drive_reset();
53     $display ("Driving the reset");
54     Clk <= 1'b0;
55     x_in <= 0;
56     @ (negedge Clk)
57     Rst = 1;
58     @ (posedge Clk)
59     Rst = 1;
60
```



```

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:

$$\begin{aligned}\mathbf{w}(0) &= \mathbf{0} \\ \mathbf{P}(0) &= \delta^{-1}\mathbf{I}\end{aligned}$$

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

```
1 module RLS (
2     input Clk, Rst,
3     input signed [15:0] x_in, // Input signal
4     output signed [15:0] y_out, // Output signal
5     output signed [15:0] w0, w1, w2, w3, // Weights
6     output signed [15:0] err, // Errors
7     output signed [15:0] p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13,
8         p14, p15, p16 // Auto correlation Matrix
9 );
10 // Here 4 bits are used for decimal and 12 bits are used for representing
11     fractions
12
13 parameter signed [15:0] gamma = 16'b0000111100111111, gamma_inverse = 16'
14     b0001000011000101 ; // gamma = 0.95
15
16 reg signed [15:0] p[0:15]; // Inverse of Auto Correlarion Function
17 wire signed [15:0] p_u[0:15], r[0:15] , p_u_raw_wire[0:15]; // Temp value
18     for storing past values of p
19 reg signed [15:0] k[0:3]; // Gain Vector
20
21 reg signed [15:0] wn[0:3], x[0:3] ;
22 wire signed [15:0] wn_u[0:3], k_1[0:3], q[0:3];
23 wire signed [15:0] m1,m2,m3,m4,l, k_scale;
24 wire signed [31:0] m11, m21, m31, m41, y_out1, p_11, p_12, p_13, p_14, p_21,
25     p_22, p_23, p_24, p_31, p_32, p_33, p_34, p_41, p_42, p_43, p_44;
26 reg signed [15:0] d_in;
27
28 FIR_Filter fir(
29     .Clk(Clk),
```



```

28     .Rst(Rst),
29     .x(x_in),
30     .d(d_in)
31 );
32
33
34 MAT_VECT_Mul M1(
35     .Mat(p),
36     .Vect(x),
37     .Vect_out(k_1)
38 );
39
40
41 VECT_VECT_Mul M2(
42     .Vect1(x),
43     .Vect2(k_1),
44     .Vect_out(l)
45 );
46
47
48 VECT_MAT_Mul M3(
49     .Mat(p),
50     .Vect(x),
51     .Vect_out(q)
52 );
53
54
55 VECT_VECT_2Mul M4(
56     .Vect1(k),
57     .Vect2(q),
58     .Mat_out(r)
59 );
60
61
62 Divider D0 (
63     .dividend(k_1[0]),
64     .divisor(k_scale),
65     .quotient(k[0])
66 );
67
68 Divider D1 (
69     .dividend(k_1[1]),
70     .divisor(k_scale),
71     .quotient(k[1])
72 );
73
74 Divider D2 (
75     .dividend(k_1[2]),
76     .divisor(k_scale),
77     .quotient(k[2])
78 );
79
80 Divider D3 (
81     .dividend(k_1[3]),
82     .divisor(k_scale),
83     .quotient(k[3])
84 );
85
86
87
88 always@(posedge Clk or posedge Rst)
89     if(Rst)
90         begin // Initializing the input x[-1] as 16'b0

```

```

91
92     x[3] <= 16'b0;
93     x[2] <= 16'b0;
94     x[1] <= 16'b0;
95     x[0] <= 16'b0;
96
97 end
98 else
99     begin // Updating the inputs
100
101         x[3] <= x[2];
102         x[2] <= x[1];
103         x[1] <= x[0];
104         x[0] <= x_in;
105
106     end
107
108
109
110 always@( posedge Clk or posedge Rst)
111     begin
112         if(Rst)
113             begin // Initializing the guess for the weight as wn[-1] as 16'b0
114
115                 wn[0] <= 16'b0;
116                 wn[1] <= 16'b0;
117                 wn[2] <= 16'b0;
118                 wn[3] <= 16'b0;
119
120             end
121         else
122             begin // Updating the weights
123
124                 wn[0] <= wn_u[0];
125                 wn[1] <= wn_u[1];
126                 wn[2] <= wn_u[2];
127                 wn[3] <= wn_u[3];
128
129             end
130         end
131
132
133
134
135 always@( posedge Clk or posedge Rst)
136     begin
137         if(Rst)
138             begin // Initializing the value for the inverse of autocorrelation as p
139                 [-1] = (1/d)I as d tends to zero
140
141                 p[0] <= 16'b0000000011111111;
142                 p[1] <= 16'b0000000000000000;
143                 p[2] <= 16'b0000000000000000;
144                 p[3] <= 16'b0000000000000000;
145                 p[4] <= 16'b0000000000000000;
146                 p[5] <= 16'b0000000011111111;
147                 p[6] <= 16'b0000000000000000;
148                 p[7] <= 16'b0000000000000000;
149                 p[8] <= 16'b0000000000000000;
150                 p[9] <= 16'b0000000000000000;
151                 p[10] <= 16'b0000000011111111;
152                 p[11] <= 16'b0000000000000000;
153                 p[12] <= 16'b0000000000000000;

```

```

153         p[13] <= 16'b0000000000000000;
154         p[14] <= 16'b0000000000000000;
155         p[15] <= 16'b0000000011111111;
156
157     end
158 else
159     begin // Updating the inverse of autocorrelation
160
161         p[0] <= p_u[0];
162         p[1] <= p_u[1];
163         p[2] <= p_u[2];
164         p[3] <= p_u[3];
165         p[4] <= p_u[4];
166         p[5] <= p_u[5];
167         p[6] <= p_u[6];
168         p[7] <= p_u[7];
169         p[8] <= p_u[8];
170         p[9] <= p_u[9];
171         p[10] <= p_u[10];
172         p[11] <= p_u[11];
173         p[12] <= p_u[12];
174         p[13] <= p_u[13];
175         p[14] <= p_u[14];
176         p[15] <= p_u[15];
177
178     end
179 end
180
181 assign y_out1 = $signed(wn[0])*$signed(x[0]) + $signed(wn[1])*$signed(x[1])
182             + $signed(wn[2])*$signed(x[2]) + $signed(wn[3])*$signed(x[3]);
183 assign y_out = y_out1[27:12];
184
185 assign err = d_in - y_out;
186
187 assign k_scale = gamma + 1;
188
189 assign m11 = $signed(err)*$signed(k[0]);
190 assign m1 = m11[27:12];
191
192 assign m21 = $signed(err)*$signed(k[1]);
193 assign m2 = m21[27:12];
194
195 assign m31 = $signed(err)*$signed(k[2]);
196 assign m3 = m31[27:12];
197
198 assign m41 = $signed(err)*$signed(k[3]);
199 assign m4 = m41[27:12];
200
201 assign wn_u[0] = wn[0] + m1;
202 assign wn_u[1] = wn[1] + m2;
203 assign wn_u[2] = wn[2] + m3;
204 assign wn_u[3] = wn[3] + m4;
205
206
207 assign w0 = wn[0];
208 assign w1 = wn[1];
209 assign w2 = wn[2];
210 assign w3 = wn[3];
211
212 assign p1 = p_u[0];
213 assign p2 = p_u[1];
214 assign p3 = p_u[2];

```

```

215 assign p4 = p_u[3];
216 assign p5 = p_u[4];
217 assign p6 = p_u[5];
218 assign p7 = p_u[6];
219 assign p8 = p_u[7];
220 assign p9 = p_u[8];
221 assign p10 = p_u[9];
222 assign p11 = p_u[10];
223 assign p12 = p_u[11];
224 assign p13 = p_u[12];
225 assign p14 = p_u[13];
226 assign p15 = p_u[14];
227 assign p16 = p_u[15];
228
229 assign p_11 = gamma_inverse * p_u_raw_wire[0];
230 assign p_u[0] = p_11[27:12];
231
232 assign p_12 = gamma_inverse * p_u_raw_wire[1];
233 assign p_u[1] = p_12[27:12];
234
235 assign p_13 = gamma_inverse * p_u_raw_wire[2];
236 assign p_u[2] = p_13[27:12];
237
238 assign p_14 = gamma_inverse * p_u_raw_wire[3];
239 assign p_u[3] = p_14[27:12];
240
241 assign p_21 = gamma_inverse * p_u_raw_wire[4];
242 assign p_u[4] = p_21[27:12];
243
244 assign p_22 = gamma_inverse * p_u_raw_wire[5];
245 assign p_u[5] = p_22[27:12];
246
247 assign p_23 = gamma_inverse * p_u_raw_wire[6];
248 assign p_u[6] = p_23[27:12];
249
250 assign p_24 = gamma_inverse * p_u_raw_wire[7];
251 assign p_u[7] = p_24[27:12];
252
253 assign p_31 = gamma_inverse * p_u_raw_wire[8];
254 assign p_u[8] = p_31[27:12];
255
256 assign p_32 = gamma_inverse * p_u_raw_wire[9];
257 assign p_u[9] = p_32[27:12];
258
259 assign p_33 = gamma_inverse * p_u_raw_wire[10];
260 assign p_u[10] = p_33[27:12];
261
262 assign p_34 = gamma_inverse * p_u_raw_wire[11];
263 assign p_u[11] = p_34[27:12];
264
265 assign p_41 = gamma_inverse * p_u_raw_wire[12];
266 assign p_u[12] = p_41[27:12];
267
268 assign p_42 = gamma_inverse * p_u_raw_wire[13];
269 assign p_u[13] = p_42[27:12];
270
271 assign p_43 = gamma_inverse * p_u_raw_wire[14];
272 assign p_u[14] = p_43[27:12];
273
274 assign p_44 = gamma_inverse * p_u_raw_wire[15];
275 assign p_u[15] = p_44[27:12];
276
277

```

```

278
279     assign p_u_raw_wire[0] = p[0] - r[0];
280     assign p_u_raw_wire[1] = p[1] - r[1];
281     assign p_u_raw_wire[2] = p[2] - r[2];
282     assign p_u_raw_wire[3] = p[3] - r[3];
283     assign p_u_raw_wire[4] = p[4] - r[4];
284     assign p_u_raw_wire[5] = p[5] - r[5];
285     assign p_u_raw_wire[6] = p[6] - r[6];
286     assign p_u_raw_wire[7] = p[7] - r[7];
287     assign p_u_raw_wire[8] = p[8] - r[8];
288     assign p_u_raw_wire[9] = p[9] - r[9];
289     assign p_u_raw_wire[10] = p[10] - r[10];
290     assign p_u_raw_wire[11] = p[11] - r[11];
291     assign p_u_raw_wire[12] = p[12] - r[12];
292     assign p_u_raw_wire[13] = p[13] - r[13];
293     assign p_u_raw_wire[14] = p[14] - r[14];
294     assign p_u_raw_wire[15] = p[15] - r[15];
295
296
297
298 endmodule
299
300
301
302
303
304
305 // This is a Multiplier for a 4*4 Matrix to a 4*1 Vector
306 module MAT_VECT_Mul(
307     input signed [15:0] Mat[0:15],
308     input signed [15:0] Vect[0:3],
309     output signed [15:0] Vect_out[0:3]
310 );
311     wire signed [31:0] Vect_out1, Vect_out2, Vect_out3, Vect_out4;
312
313     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]);
314     assign Vect_out[0] = Vect_out1[27:12];
315     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]);
316     assign Vect_out[1] = Vect_out2[27:12];
317     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]);
318     assign Vect_out[2] = Vect_out3[27:12];
319     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]);
320     assign Vect_out[3] = Vect_out4[27:12];
321
322 endmodule
323
324
325
326
327
328
329 // This is a Multiplier for a 1*4 Vector to a 4*4 Matrix
330 module VECT_MAT_Mul(
331     input signed [15:0] Mat[0:15],
332     input signed [15:0] Vect[0:3],

```

```

333     output signed [15:0] Vect_out[0:3]
334 );
335 wire signed [31:0] Vect_out1, Vect_out2, Vect_out3, Vect_out4;
336
337 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]);
338 assign Vect_out[0] = Vect_out1[27:12];
339 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]);
340 assign Vect_out[1] = Vect_out2[27:12];
341 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]);
342 assign Vect_out[2] = Vect_out3[27:12];
343 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]);
344 assign Vect_out[3] = Vect_out4[27:12];
345
346 endmodule
347
348
349
350
351
352
353
354
355 // This is a Multiplier for a 1*4 Vector to a 4*1 Vector
356 module VECT_VECT_Mul(
357     input signed [15:0] Vect1[0:3],
358     input signed [15:0] Vect2[0:3],
359     output signed [15:0] Vect_out
360 );
361 wire signed [31:0] Vect_out1;
362
363 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]);
364 assign Vect_out = Vect_out1[27:12];
365
366 endmodule
367
368
369
370
371
372
373
374 // This is a Multiplier for a 4*1 Vector to a 1*4 Vector
375 module VECT_VECT_2Mul(
376     input signed [15:0] Vect1[0:3],
377     input signed [15:0] Vect2[0:3],
378     output signed [15:0] Mat_out[0:15]
379 );
380 wire signed [31:0] Mat_out1[0:15];
381
382 assign Mat_out1[0] = $signed(Vect1[0])*$signed(Vect2[0]);
383 assign Mat_out[0] = Mat_out1[0][27:12];
384 assign Mat_out1[1] = $signed(Vect1[0])*$signed(Vect2[1]);
385 assign Mat_out[1] = Mat_out1[1][27:12];

```

```

386 assign Mat_out1[2] = $signed(Vect1[0])*$signed(Vect2[2]);
387 assign Mat_out[2] = Mat_out1[2][27:12];
388 assign Mat_out1[3] = $signed(Vect1[0])*$signed(Vect2[3]);
389 assign Mat_out[3] = Mat_out1[3][27:12];
390 assign Mat_out1[4] = $signed(Vect1[1])*$signed(Vect2[0]);
391 assign Mat_out[4] = Mat_out1[4][27:12];
392 assign Mat_out1[5] = $signed(Vect1[1])*$signed(Vect2[1]);
393 assign Mat_out[5] = Mat_out1[5][27:12];
394 assign Mat_out1[6] = $signed(Vect1[1])*$signed(Vect2[2]);
395 assign Mat_out[6] = Mat_out1[6][27:12];
396 assign Mat_out1[7] = $signed(Vect1[1])*$signed(Vect2[3]);
397 assign Mat_out[7] = Mat_out1[7][27:12];
398 assign Mat_out1[8] = $signed(Vect1[2])*$signed(Vect2[0]);
399 assign Mat_out[8] = Mat_out1[8][27:12];
400 assign Mat_out1[9] = $signed(Vect1[2])*$signed(Vect2[1]);
401 assign Mat_out[9] = Mat_out1[9][27:12];
402 assign Mat_out1[10] = $signed(Vect1[2])*$signed(Vect2[2]);
403 assign Mat_out[10] = Mat_out1[10][27:12];
404 assign Mat_out1[11] = $signed(Vect1[2])*$signed(Vect2[3]);
405 assign Mat_out[11] = Mat_out1[11][27:12];
406 assign Mat_out1[12] = $signed(Vect1[3])*$signed(Vect2[0]);
407 assign Mat_out[12] = Mat_out1[12][27:12];
408 assign Mat_out1[13] = $signed(Vect1[3])*$signed(Vect2[1]);
409 assign Mat_out[13] = Mat_out1[13][27:12];
410 assign Mat_out1[14] = $signed(Vect1[3])*$signed(Vect2[2]);
411 assign Mat_out[14] = Mat_out1[14][27:12];
412 assign Mat_out1[15] = $signed(Vect1[3])*$signed(Vect2[3]);
413 assign Mat_out[15] = Mat_out1[15][27:12];
414
415 endmodule
416
417
418
419
420
421 // Divider for Signed bit fixed point numbers
422
423 module Divider(
424     input signed [15:0] dividend,
425     input signed [15:0] divisor,
426     output signed [15:0] quotient
427 );
428     reg signed [31:0] scaled_dividend;
429     reg signed [15:0] result;
430
431     always @(*) begin
432         if (divisor == 0) begin
433             result = 16'b0111111111111111; // Max positive value to indicate
434                 error
435         end else begin
436             scaled_dividend = dividend <<< 12;
437             result = scaled_dividend / divisor;
438         end
439     end
440
441     assign quotient = result;
442 endmodule
443
444
445
446
447 // FIR Filter Module

```

```

448 module FIR_Filter (
449     input Clk,
450     input Rst,
451     input signed [15:0] x,
452     output signed [15:0] d
453 );
454
455 reg signed [15:0] w0, w1, w2, w3;
456 reg signed [15:0] xn_0, xn_1, xn_2, xn_3;
457 reg signed [31:0] d1;
458
459
460 always @(posedge Clk)
461 begin
462     if (Rst)
463     begin
464         xn_0 <= 0;
465         xn_1 <= 0;
466         xn_2 <= 0;
467         xn_3 <= 0;
468         d1 <= 0;
469     end
470     else
471     begin
472         xn_3 <= xn_2;
473         xn_2 <= xn_1;
474         xn_1 <= xn_0;
475         xn_0 <= x;
476         d1 <= w0*xn_0 + w1*xn_1 + w2*xn_2 + w3*xn_3;
477     end
478 end
479
480 assign d = d1[27:12];
481
482 initial
483 begin
484     w0 = 16'b000010000000000000;
485     w1 = 16'b000010000000000000;
486     w2 = 16'b000010000000000000;
487     w3 = 16'b000010000000000000;
488 end
489
490 endmodule

```

Results

Following is the test bench used for LMS algorithm

```

1 module RLS_TB;
2     reg Clk, Rst;
3     reg signed [15:0] x_in;
4     reg signed [15:0] x_gen;
5     wire signed [15:0] y_out;
6     reg [15:0] input_data[0:100];
7     wire signed [15:0] d, err, w0, w1, w2, w3;
8     integer o;
9     wire signed [15:0] p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13,
        p14, p15, p16;
10
11     localparam SF = 2.0**-12.0;
12
13     RLS dut(
14         .Clk(Clk),

```



```

15     .Rst(Rst),
16     .x_in(x_in),
17     .w0(w0),
18     .w1(w1),
19     .w2(w2),
20     .w3(w3),
21     .y_out(y_out),
22     .err(err),
23     .p1(p1),
24     .p2(p2),
25     .p3(p3),
26     .p4(p4),
27     .p5(p5),
28     .p6(p6),
29     .p7(p7),
30     .p8(p8),
31     .p9(p9),
32     .p10(p10),
33     .p11(p11),
34     .p12(p12),
35     .p13(p13),
36     .p14(p14),
37     .p15(p15),
38     .p16(p16)
39 );
40
41 FIR_Filter F1(
42     .Clk(Clk),
43     .Rst(Rst),
44     .x(x_in),
45     .d(d)
46 );
47
48 always begin
49     #10 Clk = ~Clk;
50 end
51
52 initial begin
53     drive_reset();
54
55     $readmemb("RLS_inputs.txt", input_data);
56
57     for( o=0; o<100;o=o+1)
58         begin
59             drive_input(input_data[o]);
60             check_output();
61         end
62
63     repeat(30)@(negedge Clk)
64         $finish;
65 end
66
67
68 task drive_reset();
69 $display ("Driving the reset");
70 Clk <= 1'b0;
71 x_in<= 0;
72 @ (negedge Clk)
73 Rst = 1;
74 @ (posedge Clk)
75 Rst = 1;
76 @ (negedge Clk)
77 Rst = 0;

```

```

78   endtask
79
80   task drive_input(input [15:0] x_gen);
81       $display ("Recieved the ready signal and driving the input");
82       @ (negedge Clk)
83       x_in = x_gen;
84       $display(x_in*SF);
85       $display ($itor(x_in*SF));
86   endtask
87
88   task check_output();
89       $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);
90
91   endtask
92
93   initial begin
94       $dumpfile("dump_rls.vcd");
95       $dumpvars;
96   end
97
98 endmodule

```

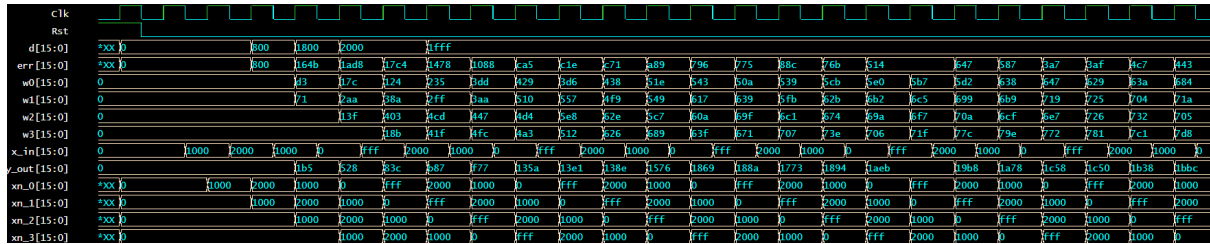


Figure 4: Timing Diagram of RLS algorithm

```

# 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