EE3510 RTL Design & Verification ICA

ICA - Independent Component Analysis
Goal - Work on RTL Verification of EMG (Electromyography) ICA
The total parts are EMD (Empirical Mode Decryption), ICA (Independent Component Analysis), and K-Means.

Team-mates:
Devansh Srivatsava (EE22BTECH11207), Gargi Behera (EE22BTECH11208), Rayi Giri Varshini (EE22BTECH11215), Talasu Sowmith (EE22BTECH11218)

In ICA, the intro part is written in the PPT, which covers what ICA is, Preprocessing techniques, ICA Algorithms, and Applications: PPT
Right now, we are working on FastICA, Grahmschimdth Orthogonalisation, and kurtosis.

Work on Normalization of signal, which is a part of preprocessing (whitening)
What is Normalization?
Normalization is a major part of whitening in the preprocessing of a signal to send it to the ICA Algorithm, which ensures the input signals follow the statistical properties that facilitate the extraction of independent components.
Whitening is the process of transforming the random variables to have unit variance and become uncorrelated (Cov(z) = I, Cor(z1, z2, z3..) = O).
Normalization does scaling and makes it a unit variant. Normalization ensures that in the whitened data, each component has the same scale (unit norm) and ensures each component has unit variance after transformation.
Normalization makes the algorithm converge fast, improves stability, and enhances performance.

$$\underline{\mathbf{w}}_{\mathbf{i}}^{\text{new}} = \frac{\mathbf{w}_{\mathbf{i}}^{\text{new}}}{\left\| \mathbf{w}_{\mathbf{i}}^{\text{new}} \right\|}.$$

where $k \in [3, n]$. The normalized vector $\underline{\mathbf{w}}_{\mathbf{i}}$ of an nD vector $\mathbf{w}_{\mathbf{i}}^{\text{new}}$ can now be obtained as follows [29]:

$$\underline{w}_{i,k}^{\text{new}} = \begin{cases} \text{Rot}_y\left(0, R_x^{3\text{D}}, V_\theta^{2\text{D}}\right), & \text{if } k = 1 \\ \text{Rot}_x\left(0, R_x^{3\text{D}}, V_\theta^{2\text{D}}\right), & \text{if } k = 2 \\ \text{Rot}_y\left(0, R_x^{(k+1)\text{D}}, V_\theta^{k\text{D}}\right), & \text{if } k \in [3, n-1] \\ \text{Rot}_y\left(0, 1, V_\theta^{n\text{D}}\right), & \text{if } k = n. \end{cases} \quad (12)$$

Finally, the estimation step in (2) was expressed in terms of CORDIC operations by replacing $\underline{\mathbf{w}}_{\mathbf{i}}$ in (9) with the converged ith weight vector $\underline{\mathbf{w}}_{\mathbf{i}}^{c}$. The estimation of the ith IC $\mathbf{s}_{\mathbf{i}}^{\text{est}} = \{s_{i,j}^{\text{est}}\}\ (j \in [1, L])$ is given by [29]

$$s_{i,j}^{\text{est}} = \text{Rot}_x\left(z_{n,j}, R_{x,j}^{(n-1)\text{D}}, \text{Vec}_\theta\left(\underline{w}_{i,n}^{c}, V_x^{(n-1)\text{D}}\right)\right). \quad (13)$$

We use CORDIC, as it doesn't have any multiplication blocks which lead to delay, made of only adders and subtractors.

Code Implementation for input as digital numbers,

```
module normalize(input [3:0] x, input clk, input reset, output reg [3:0] y);

	reg [7:0] norm_sq;
	reg [3:0] norm;
	always @(posedge clk or posedge reset) begin //assuming active high and positive edge of clk
		if(reset == 1) begin
			norm <= 0;
			norm_sq <= 0;
			y <= 0;
		end
		else begin
			norm_sq <= x*x;
			norm <= $sqrt(norm_sq);
			if(norm != 0) begin
				y <= x*4'hF/norm;
			end
			else begin
				y <= 0;
			end
		end
	end
	endmodule
```

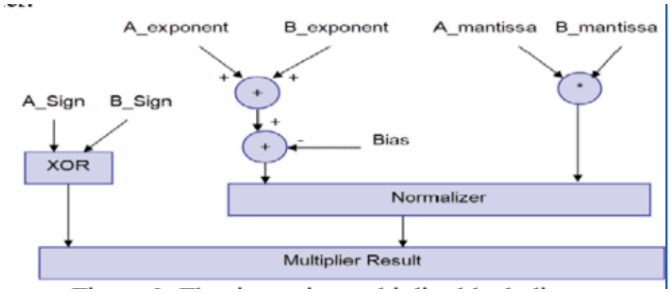
Though theoretically, it looks correct; upon adding test benches, the output is coming out to be wrong. This is due to the sqrt function. Verilog doesn't support the exact square root function, so we need to work on it.

```
module normalize_tb;
    reg [3:0] x;
    reg clk;
    reg reset;
    wire [3:0] y;

    // Instantiation
    normalize test_cases (
        .x(x),
        .clk(clk),
        .reset(reset),
        .y(y)
    );

    always #5 clk = ~clk; //clk=10ns

    initial begin
        clk = 0;
        reset = 1;
        x = 4'd0;

        // Reset after 1 full clock
        #10 reset = 0;

        x = 4'd1; #10;  //test cases
        x = 4'd2; #10;
        x = 4'd3; #10;
        x = 4'd4; #10;
        x = 4'd8; #10;
        x = 4'd0; #10;

        #10;
        $finish;
    end

    initial begin
        $monitor("At time %t: x = %d, y = %d", $time, x, y);
    end

endmodule
```

Outputs:

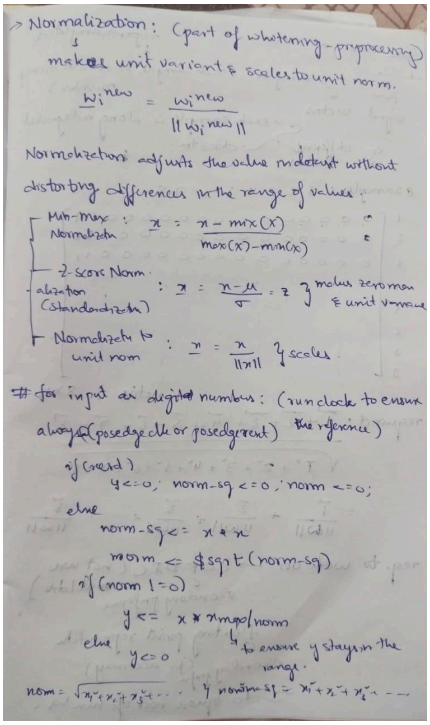
```
girivarshini@LAPTOP-H6IH7D9I:~$ vvp rtl_normalise1
At time                    0: x =  0, y =  0
At time                10000: x =  1, y =  0
At time                20000: x =  2, y =  0
At time                30000: x =  3, y =  0
At time                35000: x =  3, y = 13
At time                40000: x =  4, y = 13
At time                45000: x =  4, y =  6
At time                50000: x =  8, y =  6
At time                55000: x =  8, y =  2
At time                60000: x =  0, y =  2
At time                65000: x =  0, y =  0
```

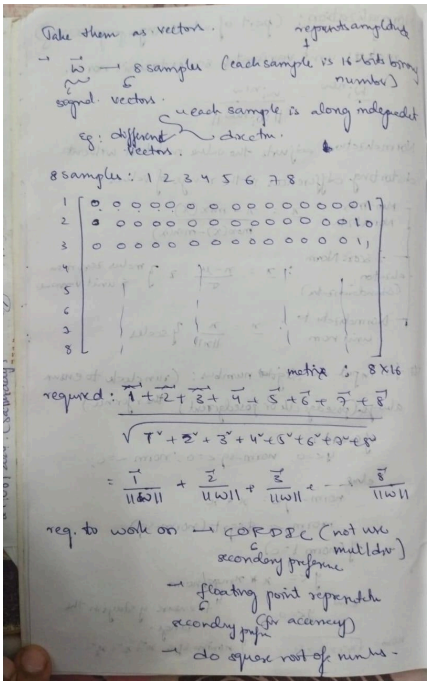
Clearly, they are wrong.
For floating point multiplier,

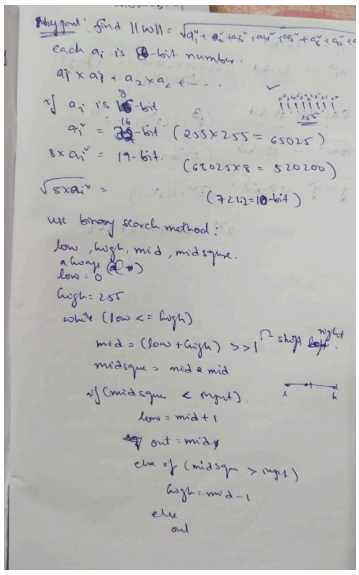


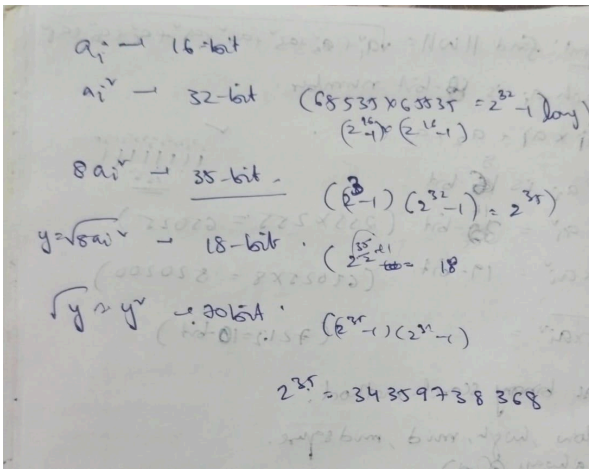
For floating point numbers, something from this: Refer

Under normalization, working on the square root function









So, we are taking 8 samples as of now, and each sample is 16-bit; we are finding the square root of the sum of squares of all 8 samples to find the norm.

In the square root of a 35-bit number (as 8*16-bit*16-bit=35-bit),
Let's do it first for binary numbers without considering the floating points, and ignoring that we are using CORDIC (i.e., we can use multiplication block also here).

Code to find square-root using binary-search algorithm without CORDIC:

```
module square_root (
    input wire [34:0] in,   //input is 35 bit (if ai is 16bit number, 8*ai*ai = 35bit)
    output reg [17:0] out  //output is log(2^35/2+1) =18)
);

    reg [17:0] low;
    reg [17:0] high;
    reg [17:0] mid;
    reg [69:0] mid_square; //35bit ka square is 70bit

    always @(*) begin
        low = 0;
        high = 185363;
        out = 0;

        while (low <= high) begin
            mid = (low + high) >> 1;
            mid_square = mid * mid;

            if (mid_square < in) begin
                low = mid + 1;
                out = mid; //approximation
            end else if (mid_square > in) begin
                high = mid - 1;
            end else begin
                out = mid; //final output
                low = high + 1; //to exit loop
            end
        end
    end
endmodule
```

Test benches:

```
module square_root_tb;
    reg [34:0] in;
    wire [17:0] out;

    // Instantiation
    square_root uut (
        .in(in),
        .out(out)
    );

    // Test vectors
    initial begin
        $display("Test square root calculation:");

        // Test Case 1: Square root of 16 (00000000000000000000000000010000)
        in = 35'd16;
        #10; // wait for the calculation
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 2: Square root of 100 (00000000000000000000000001100100)
        in = 35'd100;
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 3: Square root of 100000 (00000000000000000000000011000010010110100)
        in = 35'd100000;
        #20;
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 4: Square root of max 35-bit number
        in = 35'd34359738367; // Maximum 35-bit value
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 5: Square root of 0
        in = 35'd0;
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);
        $stop;
    end
endmodule
```

The outputs are not fully correct. It is approximated.
Problems with this code: It is giving approximated results; for 15 it is giving 3. It is taking too much time to compile for a very big number. It is not using CORDIC. We are not using a clock.

Removing the multiplication block using the shift register will make it compatible with CORDIC.
The code, testbench, and output without using multiplication block,
<saved in ubuntu>
As for binary numbers, we have inaccuracies. We are representing binary numbers in floating point and finding out the square root.
The code, testbench, and output for binary to floating point representation: <gargi has>
The code, testbench, and output for square root with floating point representation.
Square_root_copy

Codes for things:

1. For multiplication using shift registers:

```
module multiplier (
    input [17:0] a,             // First operand
    input [17:0] b,             // Second operand
    output reg [69:0] result      // Result of multiplication
);
    integer i;                  // Loop counter

    always @(*) begin
```

```
        result = 0;              // Initialize result
        for (i = 0; i < 18; i = i + 1) begin
            if (b[i]) begin
                result = result + (a << i); // Shift and add if bit is set
            end
        end
    end
endmodule
```

2. For making square roots from the binary method:

Code:

```
module square_root (
    input wire [34:0] in,   //input is 35 bit (if ai is 16bit number, 8*ai*ai = 35bit)
    output reg [17:0] out  //output is log(2^35/2+1) =18)
);

    reg [17:0] low;
    reg [17:0] high;
    reg [17:0] mid;
    reg [69:0] mid_square; //35bit ka square is 70bit

    always @(*) begin
        low = 0;
        high = 185363;
        out = 0;

        while (low <= high) begin
            mid = (low + high) >> 1;
            mid_square = mid * mid;

            if (mid_square < in) begin
                low = mid + 1;
                out = mid; //approximation
            end else if (mid_square > in) begin
                high = mid - 1;
            end else begin
                out = mid; //final output
                low = high + 1; //to exit loop
            end
        end
    end
endmodule
```

Test Bench:

```
module square_root_tb;
    reg [34:0] in;
    wire [17:0] out;

    // Instantiation
    square_root uut (
        .in(in),
        .out(out)
    );

    // Test vectors
    initial begin
        $display("Test square root calculation:");

        // Test Case 1: Square root of 16 (00000000000000000000000000010000)
        in = 35'd16;
        #10; // wait for the calculation
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 2: Square root of 100 (00000000000000000000000001100100)
        in = 35'd100;
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 3: Square root of 100000
(00000000000000000000000011000010010110100)
        in = 35'd100000;
        #20;
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 4: Square root of a max 35-bit number
        in = 35'd34359738367; // Maximum 35-bit value
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 5: Square root of 0
        in = 35'd0;
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);
        $stop;
    end
endmodule
```

3. For making square roots from the binary method without multiplication block:
Code:

```
module square_root (
    input wire [34:0] in,   // Input is 35-bit
    output reg [17:0] out   // Output is 18-bit
);

    reg [17:0] low;
    reg [17:0] high;
    reg [17:0] mid;
    reg [69:0] mid_square;   // 35-bit square is 70-bit

    // Task for multiplication using shift-and-add method
    task multiply;
        input [17:0] a;         // First operand
        input [17:0] b;         // Second operand
        output reg [69:0] result; // Result of multiplication
        integer i;              // Loop counter
        begin
            result = 0;         // Initialize result
            for (i = 0; i < 18; i = i + 1) begin
                if (b[i]) begin
                    result = result + (a << i); // Shift and add if bit is set
                end
            end
        end
    endtask

    always @(*) begin
        low = 0;
        high = 185363;          // Maximum possible square root of a 35-bit number
        out = 0;

        while (low <= high) begin
            mid = (low + high) >> 1;

            // Use the multiply task to calculate mid * mid
            multiply(mid, mid, mid_square);

            if (mid_square < in) begin
                low = mid + 1;
                out = mid; // Approximation
            end else if (mid_square > in) begin
```

```
            high = mid - 1;
        end else begin
            out = mid; // Final output
            low = high + 1; // Exit loop
        end
    end
end

endmodule
```

TestBench:

```
module square_root_tb;

    // Inputs
    reg [34:0] in;

    // Outputs
    wire [17:0] out;

    // Instantiate the Unit Under Test (UUT)
    square_root uut (
        .in(in),
        .out(out)
    );

    // Test vectors
    initial begin
        $display("Test square root calculation:");

        // Test Case 1: Square root of 16 (00000000000000000000000000010000)
        in = 35'd16;
        #10; // wait for the calculation
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 2: Square root of 100 (00000000000000000000000001100100)
        in = 35'd100;
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 3: Square root of 100000
(00000000000000000000000011000010010110100)
        in = 35'd100000;
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);
```

```
        // Test Case 4: Square root of a max 35-bit number
        in = 35'd34359738367; // Maximum 35-bit value
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Test Case 5: Square root of 0
        in = 35'd0;
        #10;
        $display("Input: %d, Calculated Square Root: %d", in, out);

        // Finish simulation
        $stop;
    end
endmodule
```

4. Floating point including the square root:
Code:

```
`default_nettype none
`timescale 1ns / 1ps

module sqrt #(
    parameter WIDTH = 8,  // width of radicand
    parameter FBITS = 0   // fractional bits (for fixed point)
    ) (
    input  wire clk,
    input  wire start,              // start signal
    output reg  busy,               // calculation in progress
    output reg  valid,              // root and rem are valid
    input  wire [WIDTH-1:0] rad,    // radicand
    output reg  [WIDTH-1:0] root,   // root
    output reg  [WIDTH-1:0] rem     // remainder
    );

    reg [WIDTH-1:0] x, x_next;      // radicand copy
    reg [WIDTH-1:0] q, q_next;      // intermediate root (quotient)
    reg [WIDTH+1:0] ac, ac_next;    // accumulator (2 bits wider)
    reg [WIDTH+1:0] test_res;       // sign test result (2 bits wider)

    localparam ITER = (WIDTH + FBITS) >> 1;  // iterations are half radicand + fbits width
    reg [$clog2(ITER)-1:0] i;       // iteration counter

    always @(*) begin
        test_res = ac - {q, 2'b01};
```

```
        if (test_res[WIDTH+1] == 0) begin  // test_res ≥0? (check MSB)
            {ac_next, x_next} = {test_res[WIDTH-1:0], x, 2'b0};
            q_next = {q[WIDTH-2:0], 1'b1};
        end else begin
            {ac_next, x_next} = {ac[WIDTH-1:0], x, 2'b0};
            q_next = q << 1;
        end
    end

    always @(posedge clk) begin
        if (start) begin
            busy <= 1;
            valid <= 0;
            i <= 0;
            q <= 0;
            {ac, x} <= {{WIDTH{1'b0}}, rad, 2'b0};
        end else if (busy) begin
            if (i == ITER - 1) begin  // we're done
                busy <= 0;
                valid <= 1;
                root <= q_next;
                rem <= ac_next[WIDTH+1:2];  // undo final shift
            end else begin  // next iteration
                i <= i + 1;
                x <= x_next;
                ac <= ac_next;
                q <= q_next;
            end
        end
    end
endmodule
```

Testbench:

```
`default_nettype none
`timescale 1ns / 1ps

module sqrt_tb;

    parameter CLK_PERIOD = 10;
    parameter WIDTH = 16;
    parameter FBITS = 8;
    parameter SF = 2.0 ** -8.0;  // Q8.8 scaling factor is 2^-8

    reg clk;
```

```
reg start;              // start signal
wire busy;              // calculation in progress
wire valid;             // root and rem are valid
reg [WIDTH-1:0] rad;   // radicand
wire [WIDTH-1:0] root; // root
wire [WIDTH-1:0] rem;  // remainder

real rad_scaled, root_scaled; // Intermediate variables for scaled radicand and root

// Instantiate the sqrt module
sqrt #(.WIDTH(WIDTH), .FBITS(FBITS)) sqrt_inst (
    .clk(clk),
    .start(start),
    .busy(busy),
    .valid(valid),
    .rad(rad),
    .root(root),
    .rem(rem)
);

always #(CLK_PERIOD / 2) clk = ~clk;

// Continuous assignments for scaled values
always @(*) begin
    rad_scaled = rad * SF;
    root_scaled = root * SF;
end

initial begin
    $monitor("\t%d:\tsqrt(%f) = %b (%f) (rem = %b) (V=%b)",
             $time, rad_scaled, root, root_scaled, rem, valid);
end

initial begin
    clk = 1;
    start = 0;
    rad = 0;

    #100    rad = 16'b1110_1000_1001_0000;  // 232.56250000
            start = 1;
    #10     start = 0;

    #120    rad = 16'b0000_0000_0100_0000;  // 0.25
            start = 1;
```

```
        #10     start = 0;

        #120    rad = 16'b0000_0010_0000_0000;  // 2.0
                start = 1;
        #10     start = 0;

        #120    $finish;
    end
Endmodule
```

Work on the math behind Gram Schmidt Orthogonalisation, CORDIC-based approach of GS, Cross-Product using CORDIC, and Also work on the code for it.
CORDIC vectoring and rotating blocks are done using doubly pipelining.