25-SUMMER SNU INTERNSHIP

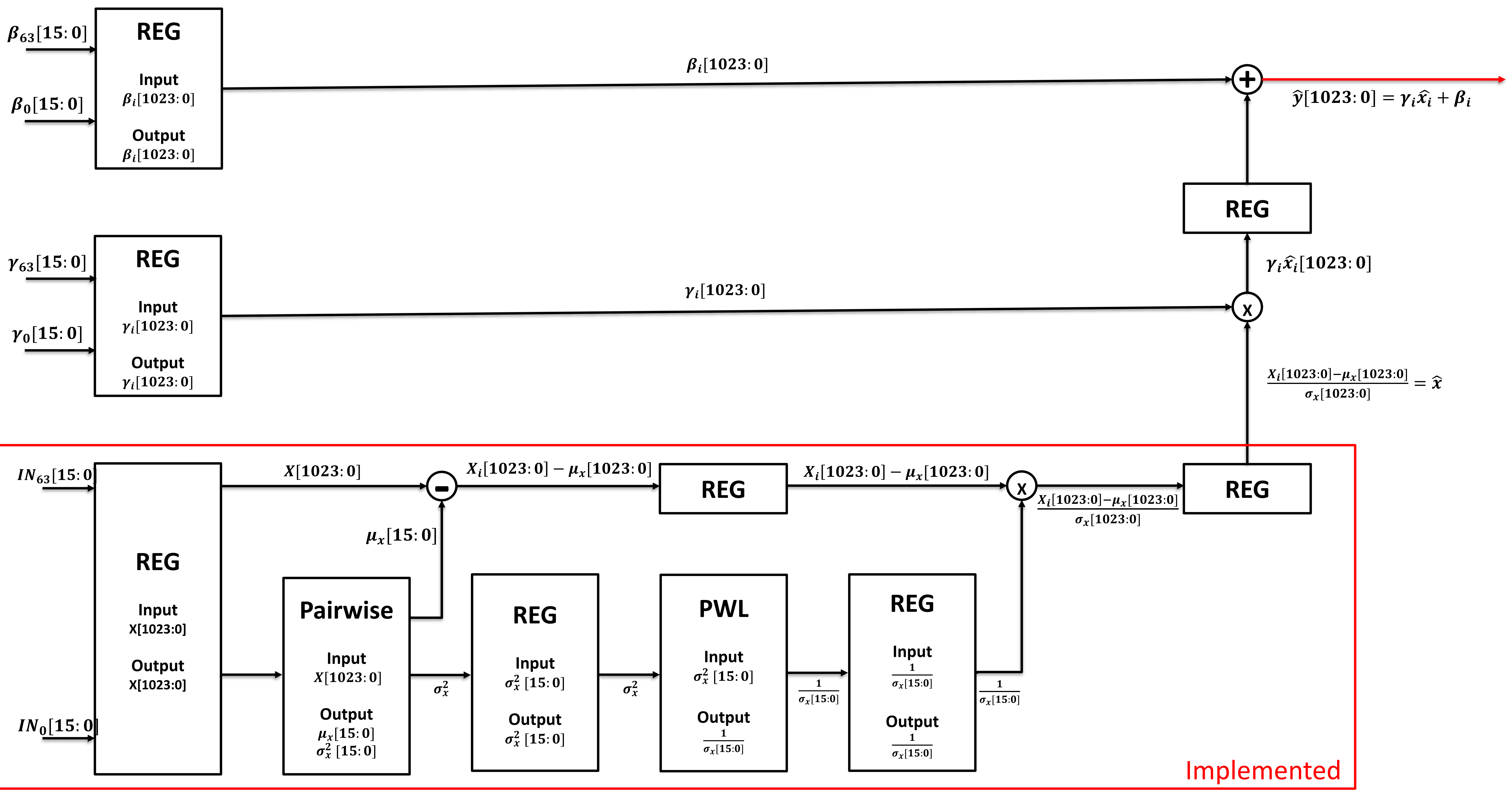
LayerNorm FPGA (Standard & Approximation)

2025.7.22.

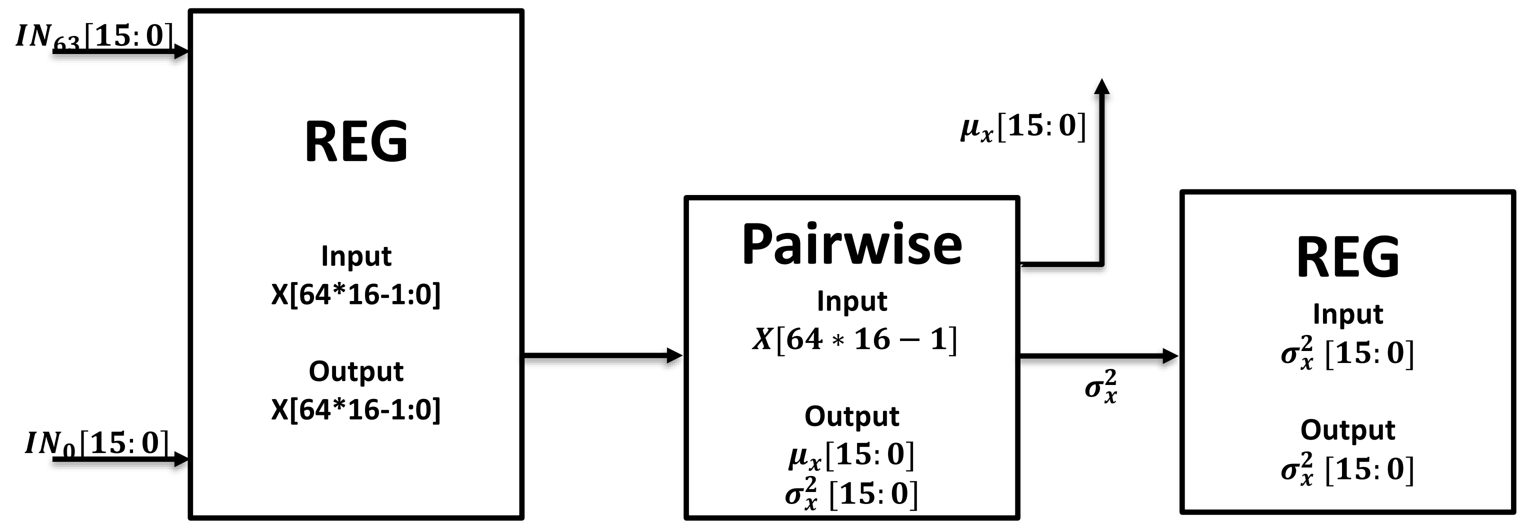
Sangyun Kim
Eunju Kim

CONTENTS

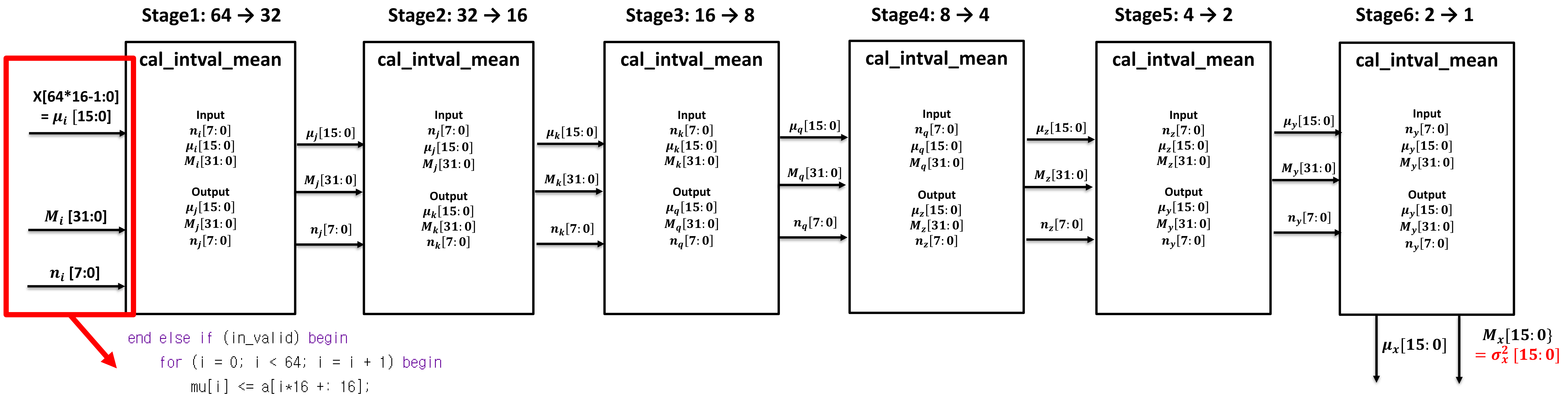
LN Approximation Model



LN Approximation Model



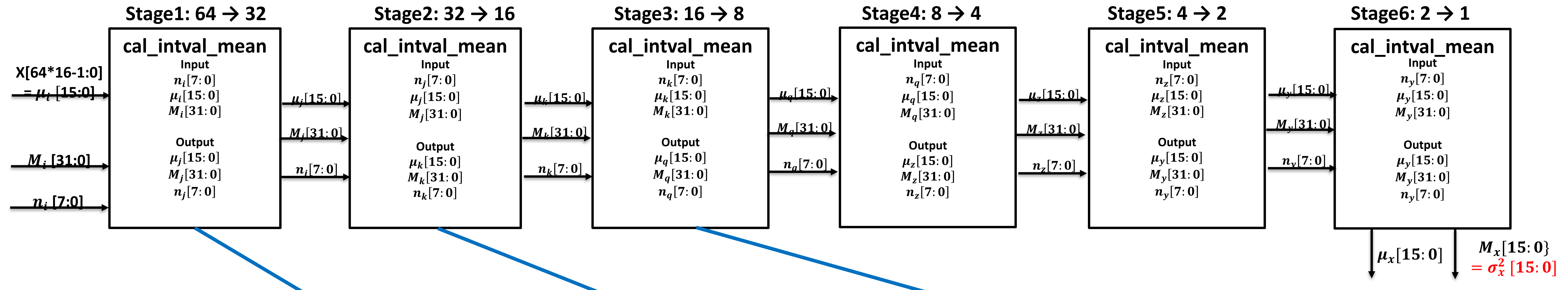
Pairwise_module



```
end else if (in_valid) begin
    for (i = 0; i < 64; i = i + 1) begin
        mu[i] <= a[i*16 +: 16];
        M[i]  <= 32'd0;
        n[i]  <= 8'd1;
    end
```

LN Approximation Model

Pairwise_module



```
module Pairwise_module (
    input  wire          clk,
    input  wire          rst_n,
    input  wire          in_valid,
    input  wire [1023:0] a,
    output reg  [15:0]  mean_out,
    output reg  [15:0]  var_out,
    output reg          out_valid
);
```

Stage1: 64 → 32

```
for (s = 0; s < 32; s = s + 1) begin : STAGE1
    cal_intval_mean u (
        .clk(clk), .rst_n(rst_n),
        .in_valid(in_valid),
        .out_valid(stage1_valid_bus[s]),
        .mu1(mu[2*s]), .mu2(mu[2*s+1]),
        .M1(M[2*s]),   .M2(M[2*s+1]),
        .n1(n[2*s]),   .n2(n[2*s+1]),
        .mu_out(mu_stage1[s]),
        .M_out(M_stage1[s]),
        .n_out(n_stage1[s])
    );
```

Stage2: 32 → 16

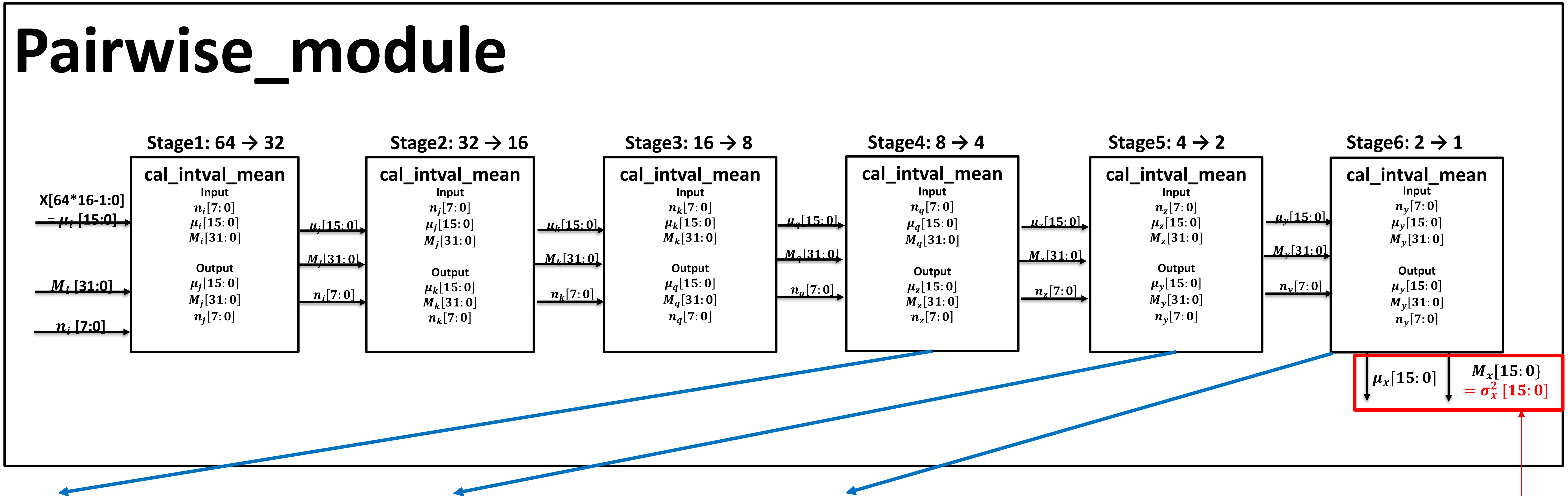
```
for (s = 0; s < 16; s = s + 1) begin : STAGE2
    cal_intval_mean u (
        .clk(clk), .rst_n(rst_n),
        .in_valid(stage1_valid),
        .out_valid(stage2_valid_bus[s]),
        .mu1(mu_stage1[2*s]), .mu2(mu_stage1[2*s+1]),
        .M1(M_stage1[2*s]),   .M2(M_stage1[2*s+1]),
        .n1(n_stage1[2*s]),   .n2(n_stage1[2*s+1]),
        .mu_out(mu_stage2[s]),
        .M_out(M_stage2[s]),
        .n_out(n_stage2[s])
    );
```

Stage3: 16 → 8

```
for (s = 0; s < 8; s = s + 1) begin : STAGE3
    cal_intval_mean u (
        .clk(clk), .rst_n(rst_n),
        .in_valid(stage2_valid),
        .out_valid(stage3_valid_bus[s]),
        .mu1(mu_stage2[2*s]), .mu2(mu_stage2[2*s+1]),
        .M1(M_stage2[2*s]),   .M2(M_stage2[2*s+1]),
        .n1(n_stage2[2*s]),   .n2(n_stage2[2*s+1]),
        .mu_out(mu_stage3[s]),
        .M_out(M_stage3[s]),
        .n_out(n_stage3[s])
    );
end
```

Stage4

LN Approximation Model


Pairwise_module
Stage1: 64 → 32
Stage2: 32 → 16
Stage3: 16 → 8
Stage4: 8 → 4
Stage5: 4 → 2
Stage6: 2 → 1
X[64*16-1:0] = μ_i[15:0]
M_i[31:0]
n_i[7:0]
cal_intval_mean Input $n_i[7:0]$ $\mu_i[15:0]$ $M_i[31:0]$ Output $\mu_j[15:0]$ $M_j[31:0]$ $n_j[7:0]$
$\mu_j[15:0]$ $M_j[31:0]$ $n_j[7:0]$
cal_intval_mean Input $n_j[7:0]$ $\mu_j[15:0]$ $M_j[31:0]$ Output $\mu_k[15:0]$ $M_k[31:0]$ $n_k[7:0]$
$\mu_k[15:0]$ $M_k[31:0]$ $n_k[7:0]$
cal_intval_mean Input $n_k[7:0]$ $\mu_k[15:0]$ $M_k[31:0]$ Output $\mu_q[15:0]$ $M_q[31:0]$ $n_q[7:0]$
$\mu_q[15:0]$ $M_q[31:0]$ $n_q[7:0]$
cal_intval_mean Input $n_q[7:0]$ $\mu_q[15:0]$ $M_q[31:0]$ Output $\mu_z[15:0]$ $M_z[31:0]$ $n_z[7:0]$
$\mu_z[15:0]$ $M_z[31:0]$ $n_z[7:0]$
cal_intval_mean Input $n_z[7:0]$ $\mu_z[15:0]$ $M_z[31:0]$ Output $\mu_y[15:0]$ $M_y[31:0]$ $n_y[7:0]$
$\mu_y[15:0]$ $M_y[31:0]$ $n_y[7:0]$
cal_intval_mean Input $n_y[7:0]$ $\mu_y[15:0]$ $M_y[31:0]$ Output $\mu_y[15:0]$ $M_y[31:0]$ $n_y[7:0]$
$\mu_x[15:0]$
$M_x[15:0\} = \sigma_x^2\ [15:0]$

Stage4: 8 → 4

```
for (s = 0; s < 4; s = s + 1) begin : STAGE4
    cal_intval_mean u (
        .clk(clk), .rst_n(rst_n),
        .in_valid(stage3_valid),
        .out_valid(stage4_valid_bus[s]),
        .mu1(mu_stage3[2*s]),  .mu2(mu_stage3[2*s+1]),
        .M1(M_stage3[2*s]),    .M2(M_stage3[2*s+1]),
        .n1(n_stage3[2*s]),    .n2(n_stage3[2*s+1]),
        .mu_out(mu_stage4[s]),
        .M_out(M_stage4[s]),
        .n_out(n_stage4[s])
    );
end
```

Stage3

Stage5: 4 → 2

```
for (s = 0; s < 2; s = s + 1) begin : STAGE5
    cal_intval_mean u (
        .clk(clk), .rst_n(rst_n),
        .in_valid(stage4_valid),
        .out_valid(stage5_valid_bus[s]),
        .mu1(mu_stage4[2*s]),  .mu2(mu_stage4[2*s+1]),
        .M1(M_stage4[2*s]),    .M2(M_stage4[2*s+1]),
        .n1(n_stage4[2*s]),    .n2(n_stage4[2*s+1]),
        .mu_out(mu_stage5[s]),
        .M_out(M_stage5[s]),
        .n_out(n_stage5[s])
    );
end
```

Stage6: 2 → 1

```
cal_intval_mean u6 (
    .clk(clk), .rst_n(rst_n),
    .in_valid(stage5_valid),
    .out_valid(stage6_valid),
    .mu1(mu_stage5[0]),  .mu2(mu_stage5[1]),
    .M1(M_stage5[0]),    .M2(M_stage5[1]),
    .n1(n_stage5[0]),    .n2(n_stage5[1]),
    .mu_out(mu_stage6),
    .M_out(M_stage6),
    .n_out()
);
```

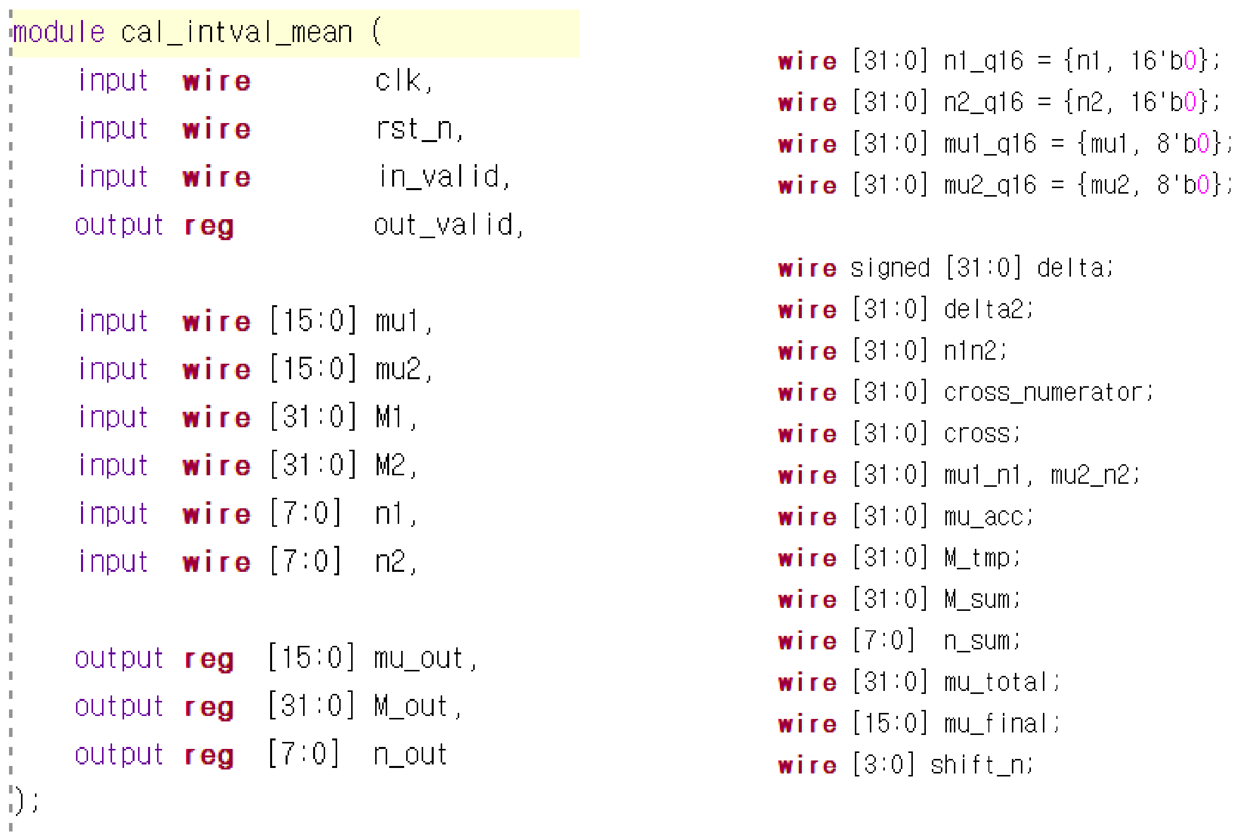
```
// Output logic
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        mean_out  <= 0;
        var_out   <= 0;
        out_valid <= 0;
    end else if (stage6_valid_d1) begin
        mean_out  <= mu_stage6;
        var_out   <= M_stage6[23:8];
        out_valid <= 1'b1;
    end else begin
        out_valid <= 1'b0;
    end
end
```

LN Approximation Model

cal_intval_mean

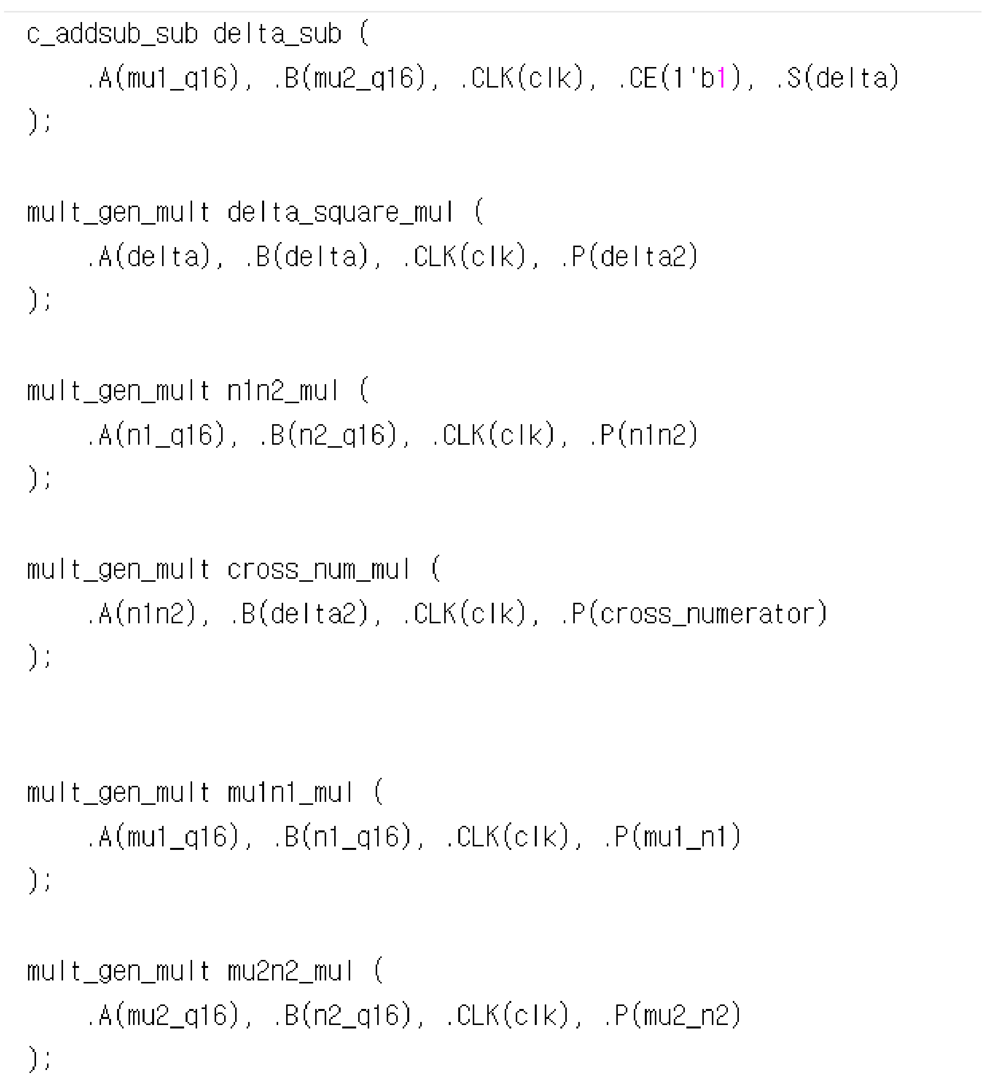
(1)	delta = mu1 - mu2	ADD
(2)	delta2 = delta * delta	MULT
(3)	cross = (n1 * n2 * delta2) / (n1 + n2)	ADD/MULT/SHIFT/LOG2
(4)	mu_acc = (mu1 * n1 + mu2 * n2)	ADD/MULT
(5)	M_out = M1 + M2 + cross	ADD
(6)	mu_out = mu_acc / (n1 + n2)	ADD/SHIFT/LOG2
(7)	n_out = n1 + n2	ADD

ADD(SUB)	c_addsub_add
MULT	mult_gen_mult
LOG2	Using LUT



```
module cal_intval_mean (
    input  wire        clk,
    input  wire        rst_n,
    input  wire        in_valid,
    output reg         out_valid,

    input  wire [15:0] mu1,
    input  wire [15:0] mu2,
    input  wire [31:0] M1,
    input  wire [31:0] M2,
    input  wire [7:0]  n1,
    input  wire [7:0]  n2,

    output reg  [15:0] mu_out,
    output reg  [31:0] M_out,
    output reg  [7:0]  n_out
);
```

```
wire [31:0] n1_q16 = {n1, 16'b0};
wire [31:0] n2_q16 = {n2, 16'b0};
wire [31:0] mu1_q16 = {mu1, 8'b0};
wire [31:0] mu2_q16 = {mu2, 8'b0};

wire signed [31:0] delta;
wire [31:0] delta2;
wire [31:0] n1n2;
wire [31:0] cross_numerator;
wire [31:0] cross;
wire [31:0] mu1_n1, mu2_n2;
wire [31:0] mu_acc;
wire [31:0] M_tmp;
wire [31:0] M_sum;
wire [7:0]  n_sum;
wire [31:0] mu_total;
wire [15:0] mu_final;
wire [3:0] shift_n;
```

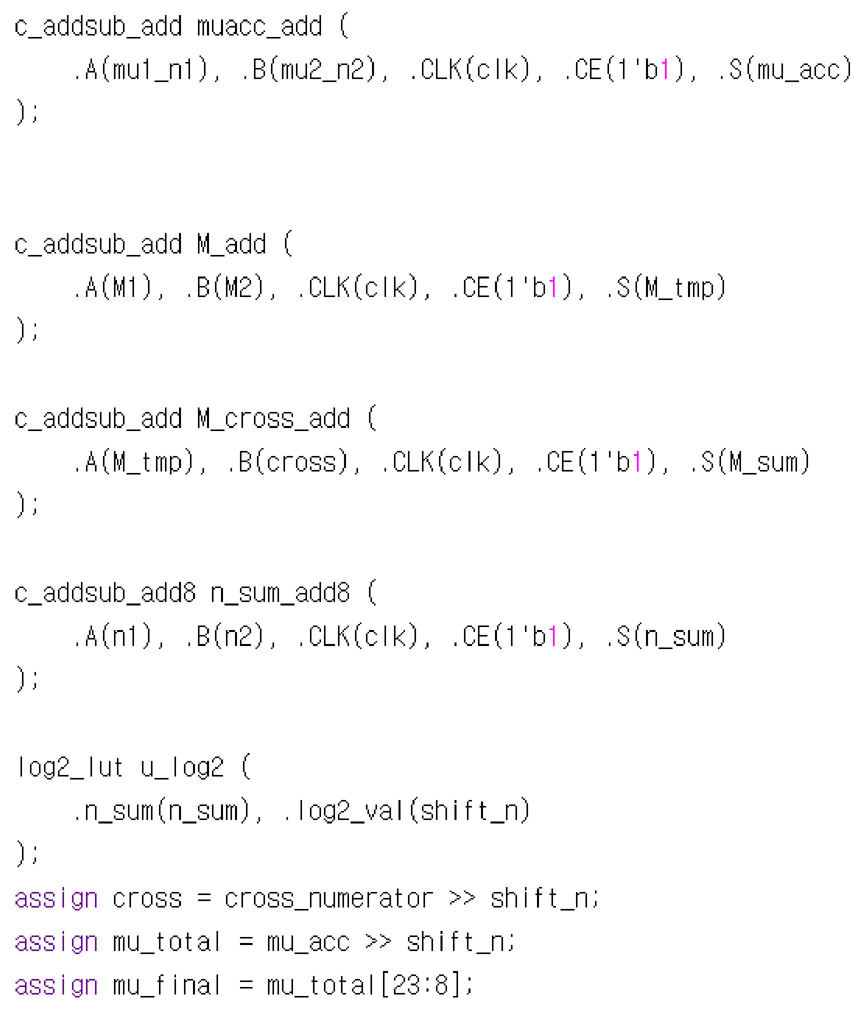


(1)
```
c_addsub_sub delta_sub (
    .A(mu1_q16), .B(mu2_q16), .CLK(clk), .CE(1'b1), .S(delta)
);
```

(2)
```
mult_gen_mult delta_square_mul (
    .A(delta), .B(delta), .CLK(clk), .P(delta2)
);
```

(3)
```
mult_gen_mult n1n2_mul (
    .A(n1_q16), .B(n2_q16), .CLK(clk), .P(n1n2)
);

mult_gen_mult cross_num_mul (
    .A(n1n2), .B(delta2), .CLK(clk), .P(cross_numerator)
);
```

(4)
```
mult_gen_mult mu1n1_mul (
    .A(mu1_q16), .B(n1_q16), .CLK(clk), .P(mu1_n1)
);

mult_gen_mult mu2n2_mul (
    .A(mu2_q16), .B(n2_q16), .CLK(clk), .P(mu2_n2)
);
```



```
c_addsub_add muacc_add (
    .A(mu1_n1), .B(mu2_n2), .CLK(clk), .CE(1'b1), .S(mu_acc)
);
```

(5)
```
c_addsub_add M_add (
    .A(M1), .B(M2), .CLK(clk), .CE(1'b1), .S(M_tmp)
);

c_addsub_add M_cross_add (
    .A(M_tmp), .B(cross), .CLK(clk), .CE(1'b1), .S(M_sum)
);
```

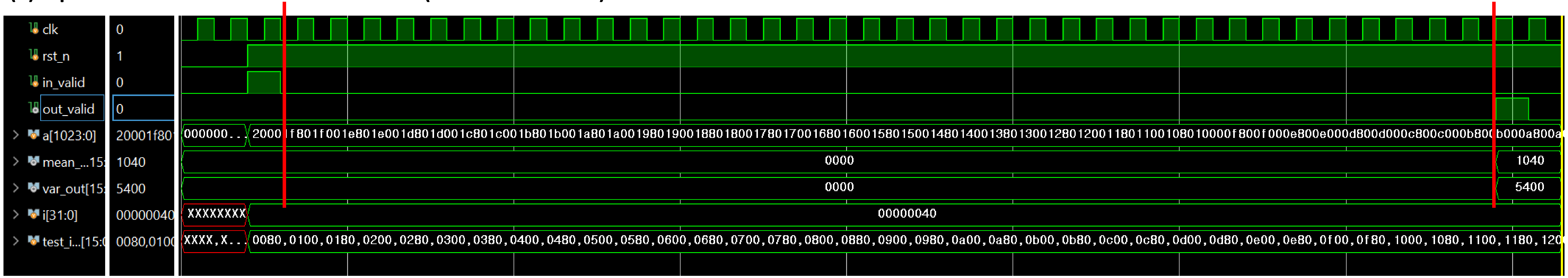
(7)
```
c_addsub_add8 n_sum_add8 (
    .A(n1), .B(n2), .CLK(clk), .CE(1'b1), .S(n_sum)
);

log2_lut u_log2 (
    .n_sum(n_sum), .log2_val(shift_n)
);
```

(3) **(6)**
```
assign cross = cross_numerator >> shift_n;
assign mu_total = mu_acc >> shift_n;
assign mu_final = mu_total[23:8];
```

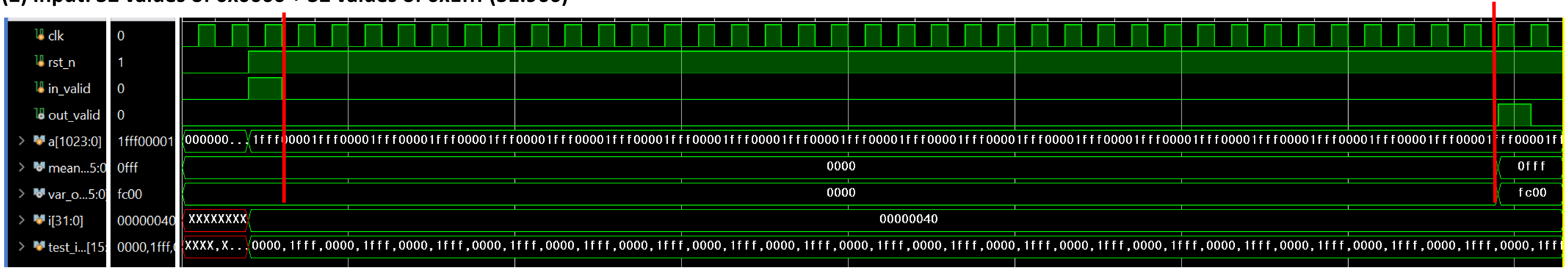
LN Approximation Model

(1) Input: 0x0080 0x0100 0x0180 … 0x2000 (0.5 1.0 1.5 … 32.0)

LATECNCY : 36

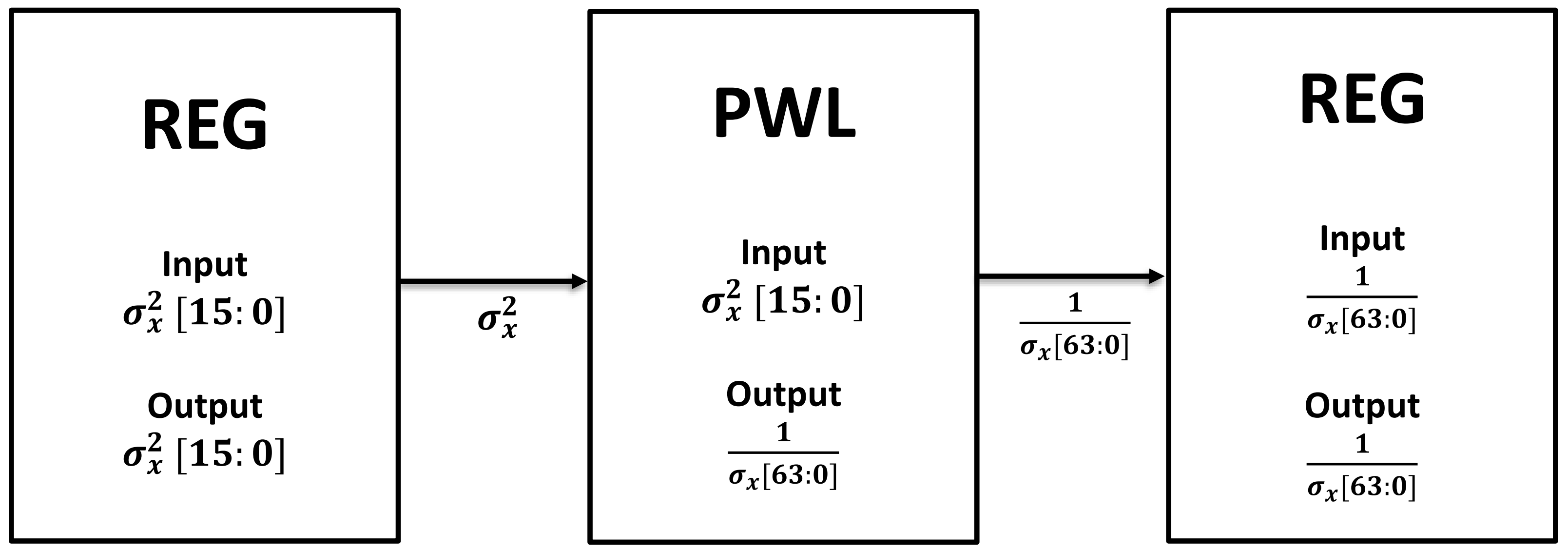

clk 0
rst_n 1
in_valid 0
out_valid 0
a[1023:0] 20001f80
mean_...15 1040
var_out[15 5400
i[31:0] 00000040
test_i...[15:0 0080,0100
000000..
2000 1f801f001e801e001d801d001c801c001b801b001a801a00198019001880180017801700168016001580150014801400138013001280120011801100108010000f800f000e800e000d800d000c800c000b800b000a800a
0000
1040
0000
5400
XXXXXXXX
00000040
XXXX,X..
0080,0100,0180,0200,0280,0300,0380,0400,0480,0500,0580,0600,0680,0700,0780,0800,0880,0900,0980,0a00,0a80,0b00,0b80,0c00,0c80,0d00,0d80,0e00,0e80,0f00,0f80,1000,1080,1100,1180,120

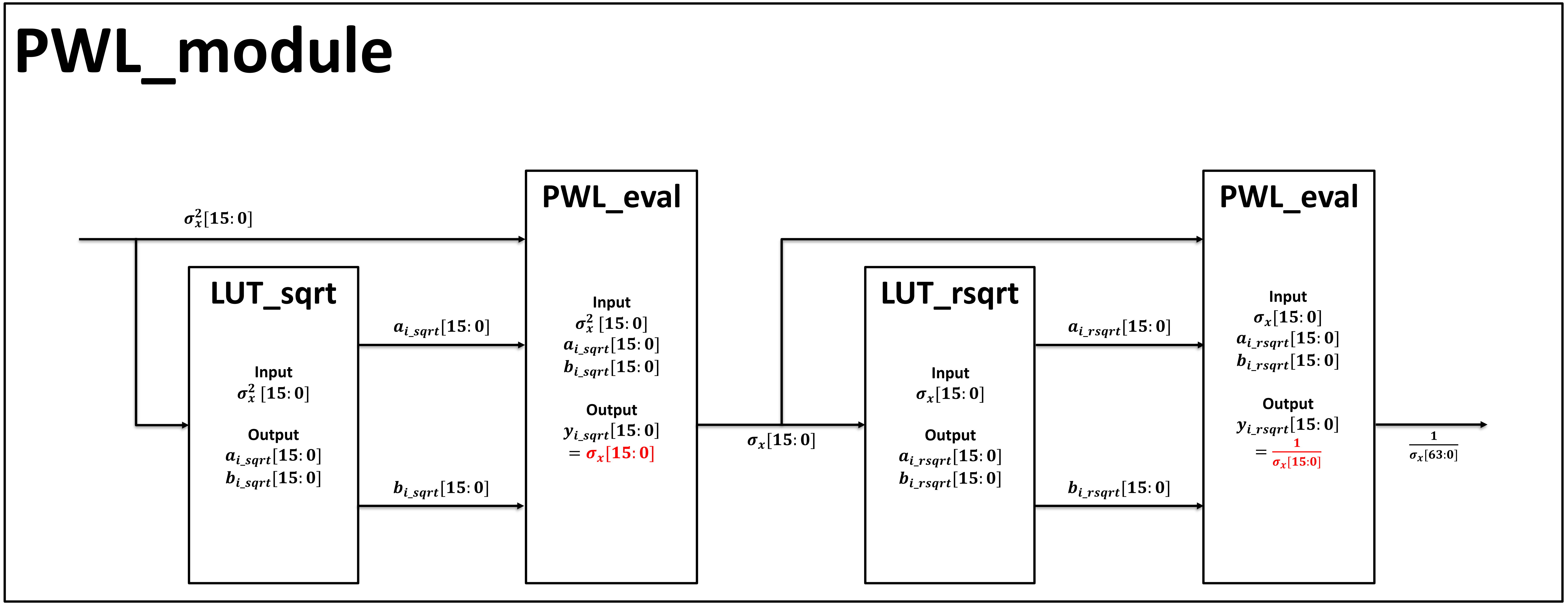
(2) Input: 32 values of 0x0000 + 32 values of 0x1fff (31.966)


clk 0
rst_n 1
in_valid 0
out_valid 0
a[1023:0] 1fff00001
mean...5:0 0fff
var_o...5:0 fc00
i[31:0] 00000040
test_i...[15 0000,1fff,
000000..
1fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001fff00001ff
0000
0fff
0000
fc00
XXXXXXXX
00000040
XXXX,X..
0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff,0000,1fff

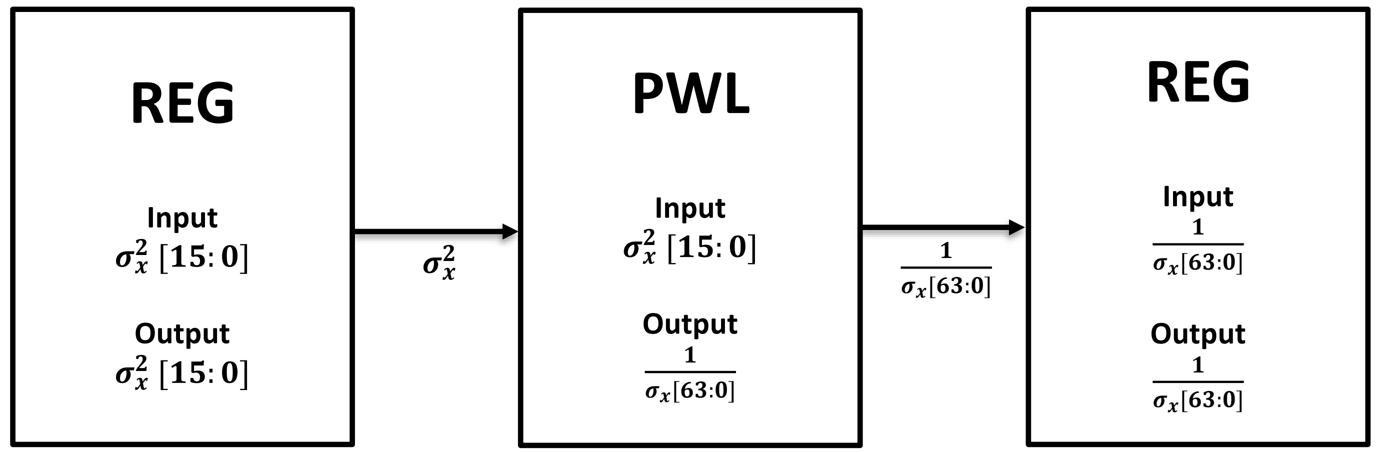
Q8.8 input hex (Dec)	(1)	(2)	Accuracy
Mean	0x1040 (16.25)	0x0fff (15.996)	100% (error=0%)
Mean expected	0x1040 (16.25)	0x0fff (15.996)	
Variance	0x5400 (84.0)	0xfc00 (252.0)	98.46% (error=1.54%)
Variance expected	0x5550 (85.313)	0xffb1 (255.246)	

LN Approximation Model

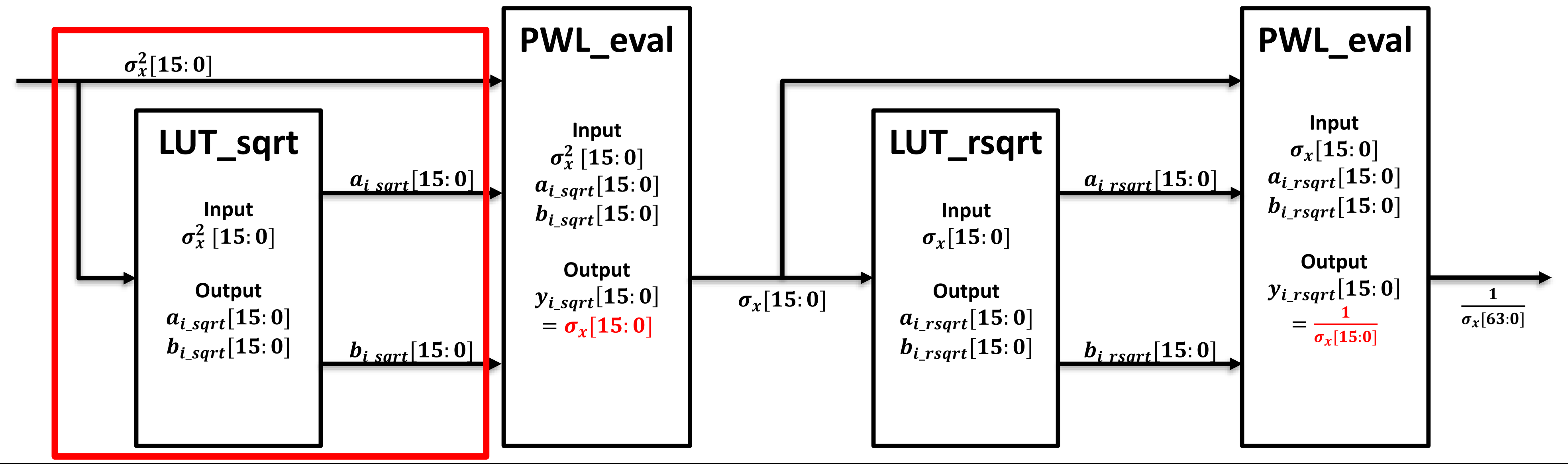




LN Approximation Model

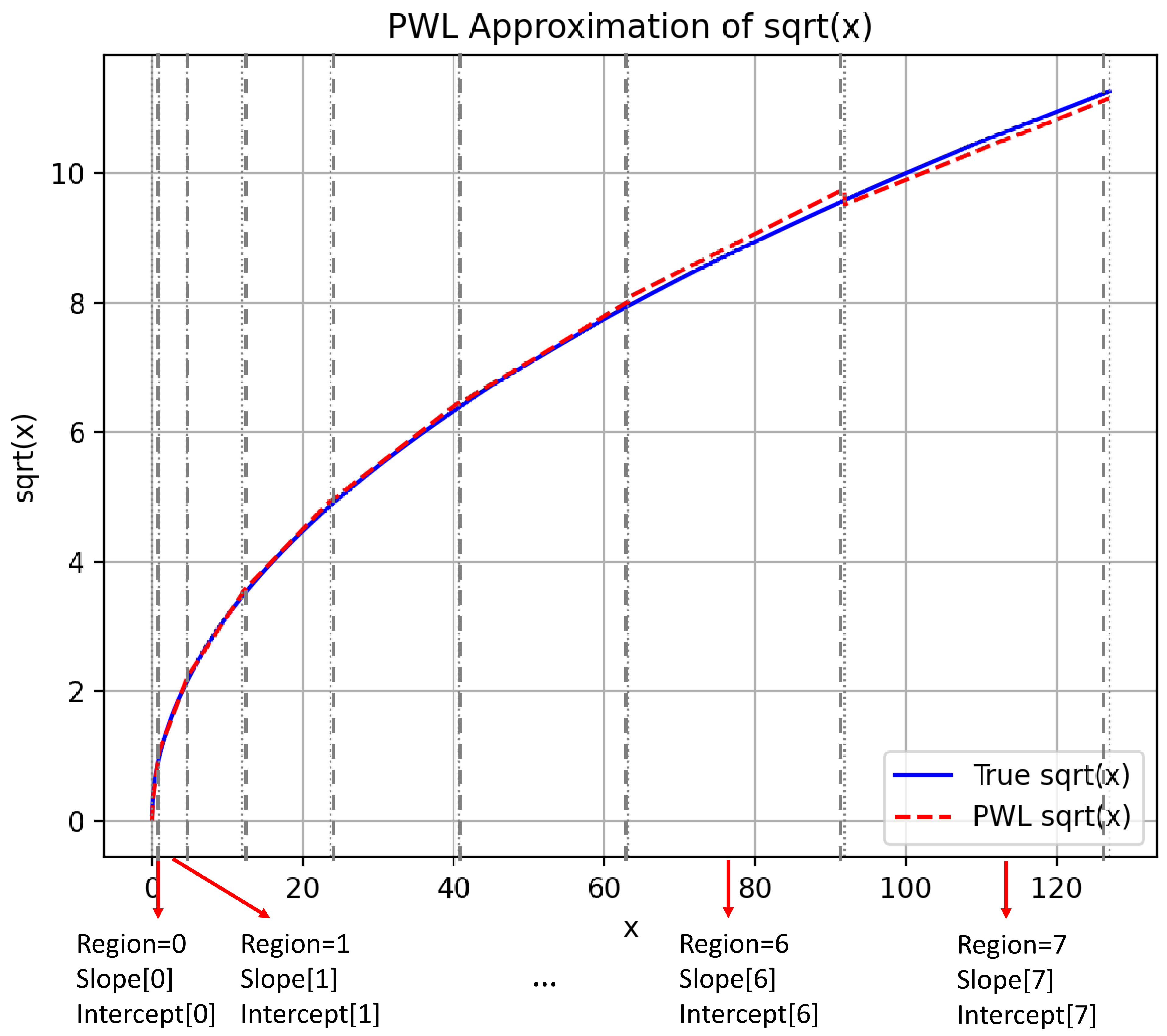


PWL_module



```verilog
module LUT_sqrt (
    input  wire        clk,
    input  wire        rst_n,
    input  wire        in_valid,
    input  wire [15:0] x_in,
    output reg         out_valid,
    output reg  [15:0] slope_out,
    output reg  [15:0] intercept_out
);
    localparam integer N_SEGMENTS = 8;

    reg [15:0] breakpoints [0:N_SEGMENTS];
    reg [15:0] slopes      [0:N_SEGMENTS-1];
    reg [15:0] intercepts  [0:N_SEGMENTS-1];

    reg [2:0] region;

    // internal wire
    reg [2:0] next_region;
    reg       region_valid;

// Region decision
always @(*) begin
    if (x_in < breakpoints[1])      next_region = 0;
    else if (x_in < breakpoints[2]) next_region = 1;
    else if (x_in < breakpoints[3]) next_region = 2;
    else if (x_in < breakpoints[4]) next_region = 3;
    else if (x_in < breakpoints[5]) next_region = 4;
    else if (x_in < breakpoints[6]) next_region = 5;
    else if (x_in < breakpoints[7]) next_region = 6;
    else                            next_region = 7;
end

// Save Region
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        region <= 0;
        region_valid <= 0;
    end else begin
        if (in_valid) begin
            region <= next_region;
            region_valid <= 1;
        end else begin
            region_valid <= 0;
        end
    end
end

// Output
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        slope_out     <= 0;
        intercept_out <= 0;
        out_valid     <= 0;
    end else begin
        slope_out     <= slopes[region];
        intercept_out <= intercepts[region];
        out_valid     <= region_valid;
    end
```

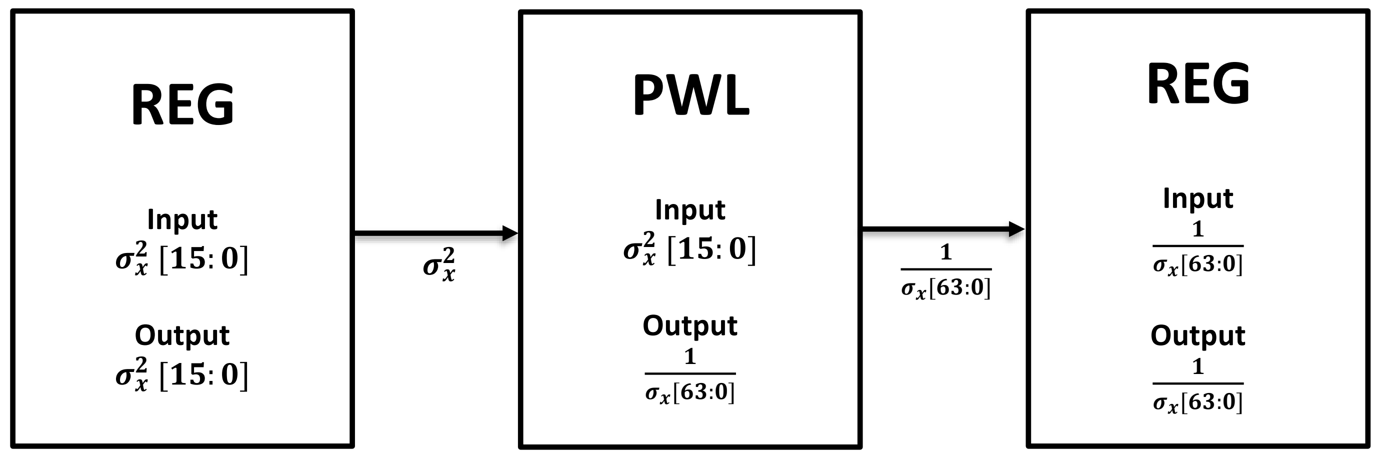
LN Approximation Model

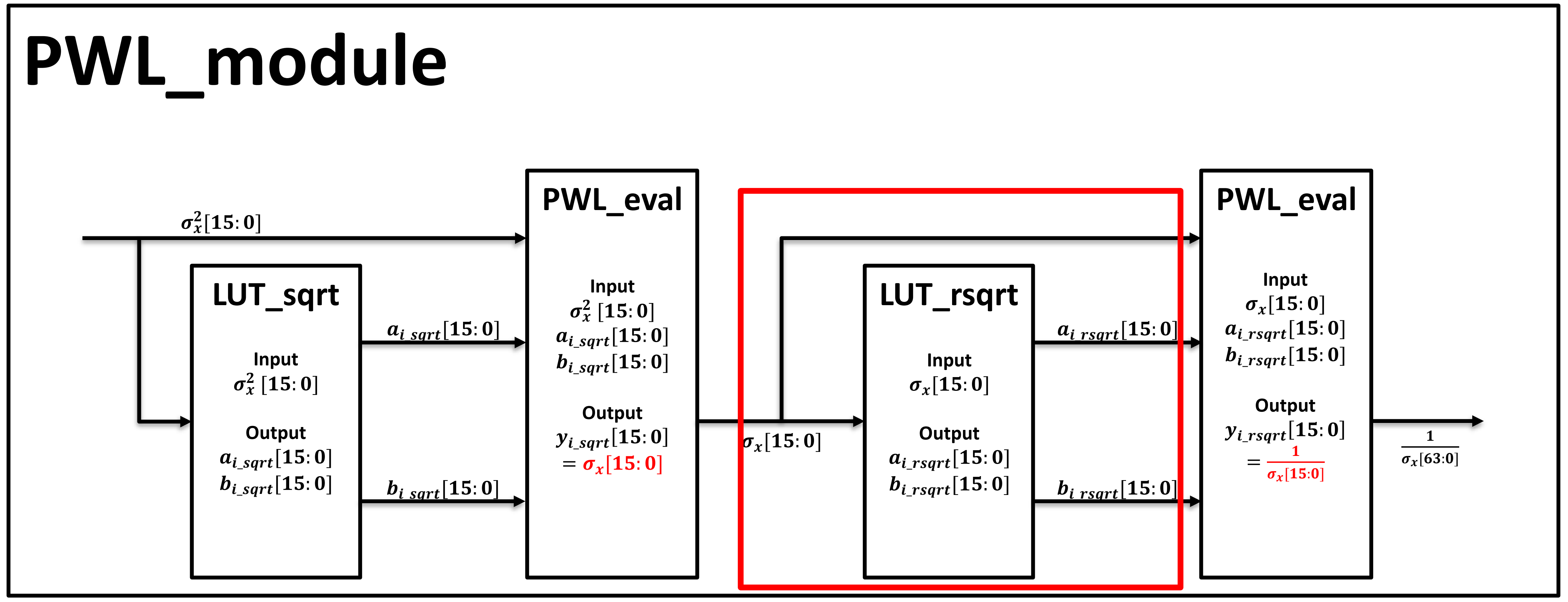


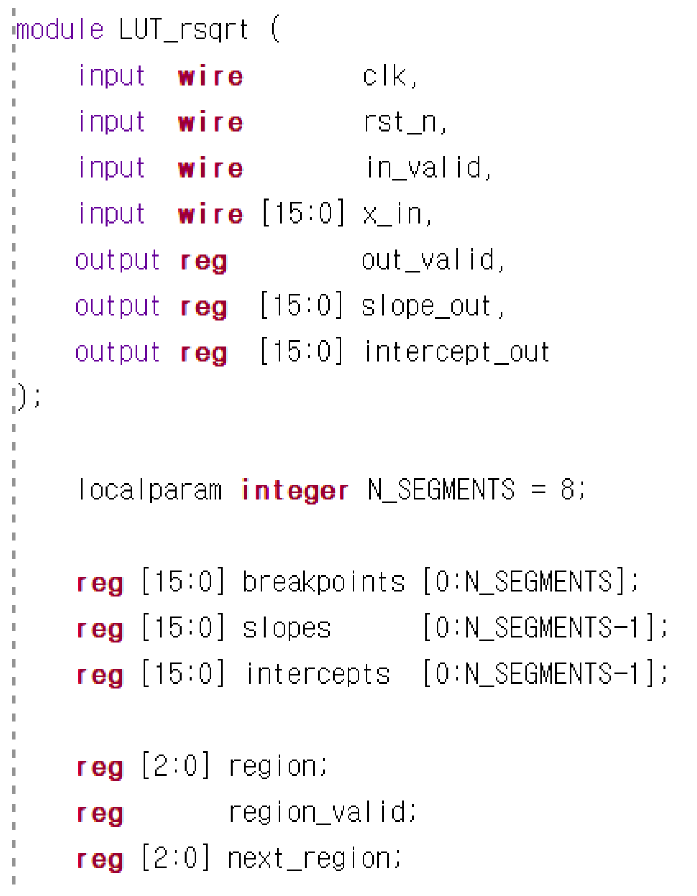
sqrt(x) = x * slopes[i] + intercepts[i]

```
initial begin
    breakpoints[0] = 16'h0003; // 0.0117
    breakpoints[1] = 16'h00F7; // 0.9648
    breakpoints[2] = 16'h04A9; // 4.6602
    breakpoints[3] = 16'h0C03; // 12.0117
    breakpoints[4] = 16'h17BF; // 23.7461
    breakpoints[5] = 16'h28A1; // 40.6328
    breakpoints[6] = 16'h3F38; // 63.2188
    breakpoints[7] = 16'h5BDB; // 91.8555
    breakpoints[8] = 16'h7F00; // 127.0000

    slopes[0] = 16'h00E2; // 0.8828
    slopes[1] = 16'h004F; // 0.3086
    slopes[2] = 16'h002D; // 0.1758
    slopes[3] = 16'h001F; // 0.1211
    slopes[4] = 16'h0017; // 0.0898
    slopes[5] = 16'h0012; // 0.0703
    slopes[6] = 16'h000F; // 0.0586
    slopes[7] = 16'h000C; // 0.0469

    intercepts[0] = 16'h0038; // 0.2188
    intercepts[1] = 16'h00C6; // 0.7734
    intercepts[2] = 16'h0166; // 1.3984
    intercepts[3] = 16'h0214; // 2.0781
    intercepts[4] = 16'h02CE; // 2.8086
    intercepts[5] = 16'h0393; // 3.5742
    intercepts[6] = 16'h0461; // 4.3789
    intercepts[7] = 16'h0536; // 5.2109
end
```

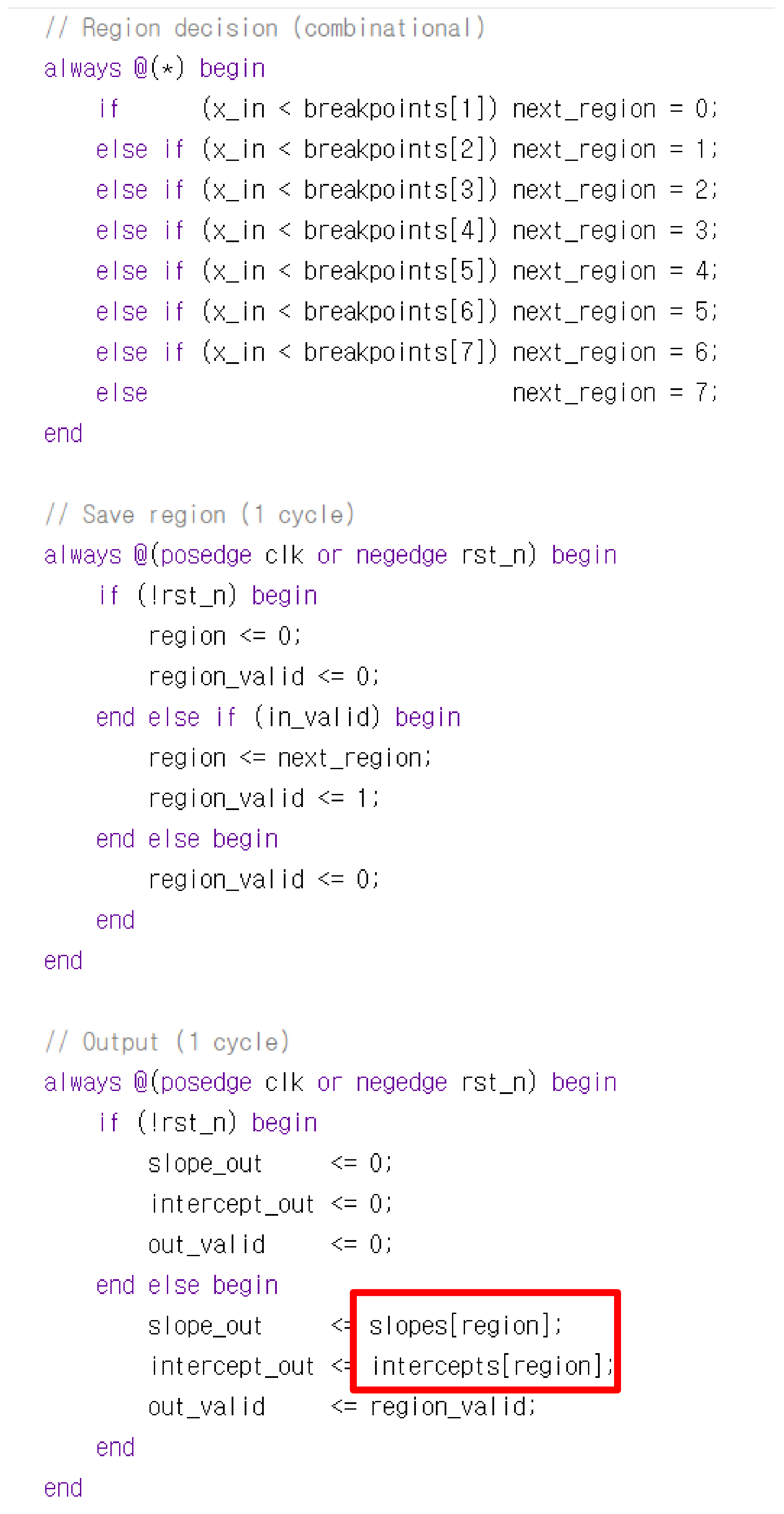
LN Approximation Model





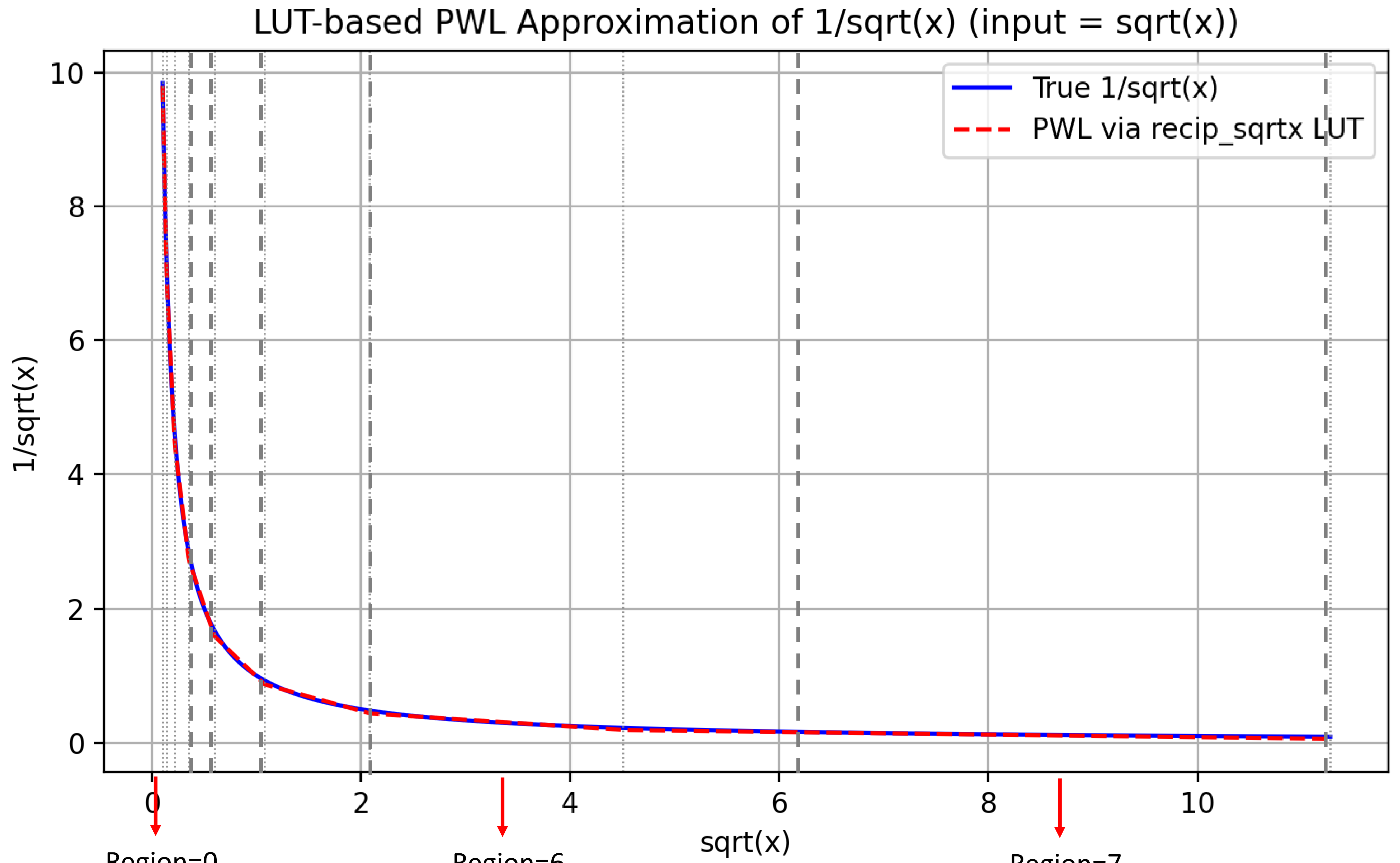


```
module LUT_rsqrt (
    input  wire        clk,
    input  wire        rst_n,
    input  wire        in_valid,
    input  wire [15:0] x_in,
    output reg         out_valid,
    output reg  [15:0] slope_out,
    output reg  [15:0] intercept_out
);

    localparam integer N_SEGMENTS = 8;

    reg [15:0] breakpoints [0:N_SEGMENTS];
    reg [15:0] slopes      [0:N_SEGMENTS-1];
    reg [15:0] intercepts  [0:N_SEGMENTS-1];

    reg [2:0] region;
    reg       region_valid;
    reg [2:0] next_region;
```



```
// Region decision (combinational)
always @(*) begin
    if      (x_in < breakpoints[1]) next_region = 0;
    else if (x_in < breakpoints[2]) next_region = 1;
    else if (x_in < breakpoints[3]) next_region = 2;
    else if (x_in < breakpoints[4]) next_region = 3;
    else if (x_in < breakpoints[5]) next_region = 4;
    else if (x_in < breakpoints[6]) next_region = 5;
    else if (x_in < breakpoints[7]) next_region = 6;
    else                            next_region = 7;
end

// Save region (1 cycle)
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        region <= 0;
        region_valid <= 0;
    end else if (in_valid) begin
        region <= next_region;
        region_valid <= 1;
    end else begin
        region_valid <= 0;
    end
end

// Output (1 cycle)
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        slope_out     <= 0;
        intercept_out <= 0;
        out_valid     <= 0;
    end else begin
        slope_out     <= slopes[region];
        intercept_out <= intercepts[region];
        out_valid     <= region_valid;
    end
end
```

LN Approximation Model



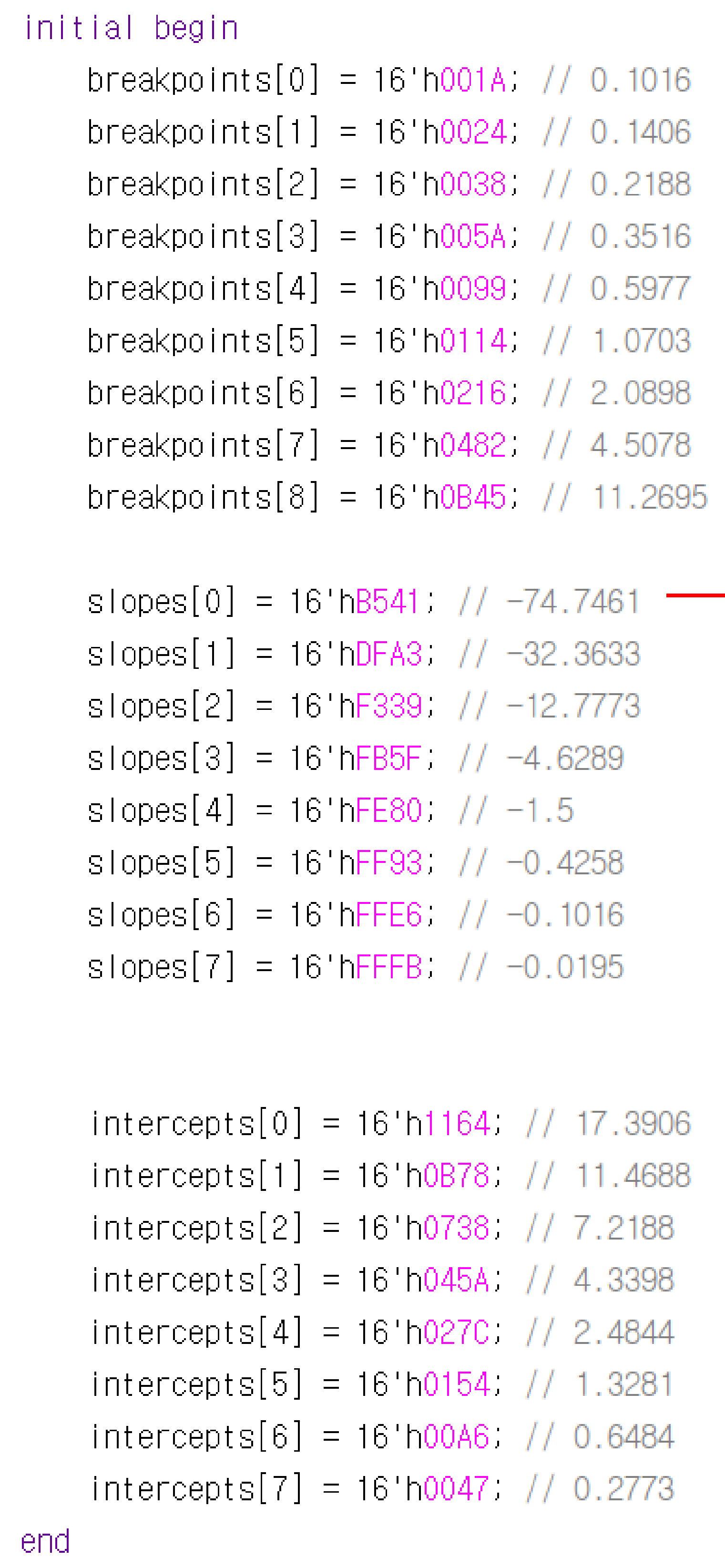
Region=0
Slope[0]
Intercept[0]

…

Region=6
Slope[6]
Intercept[6]

Region=7
Slope[7]
Intercept[7]

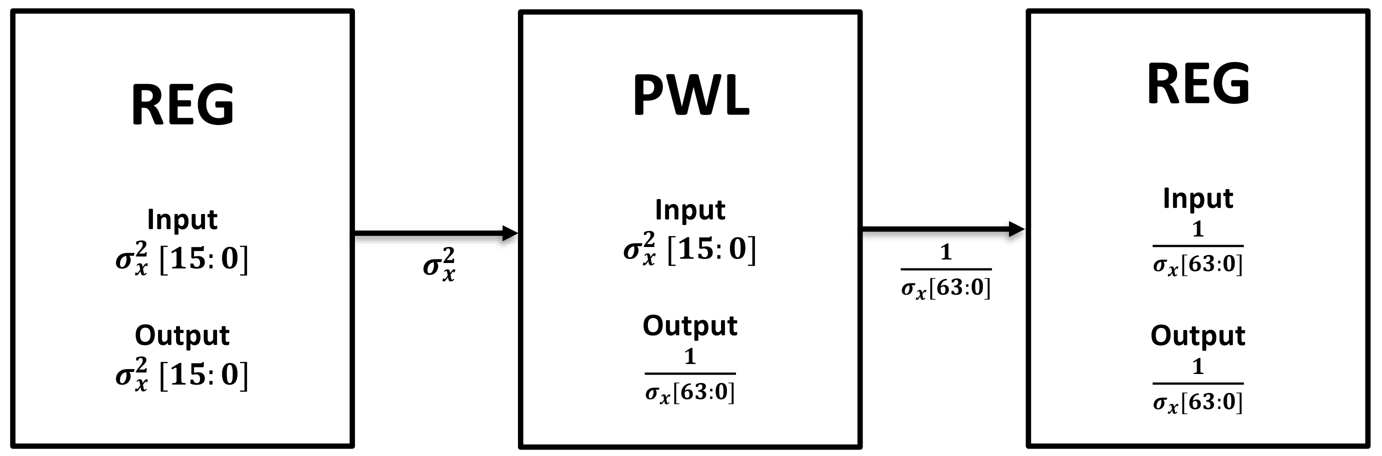
1/sqrt(x) = sqrt(x) * slopes[i] + intercepts[i]



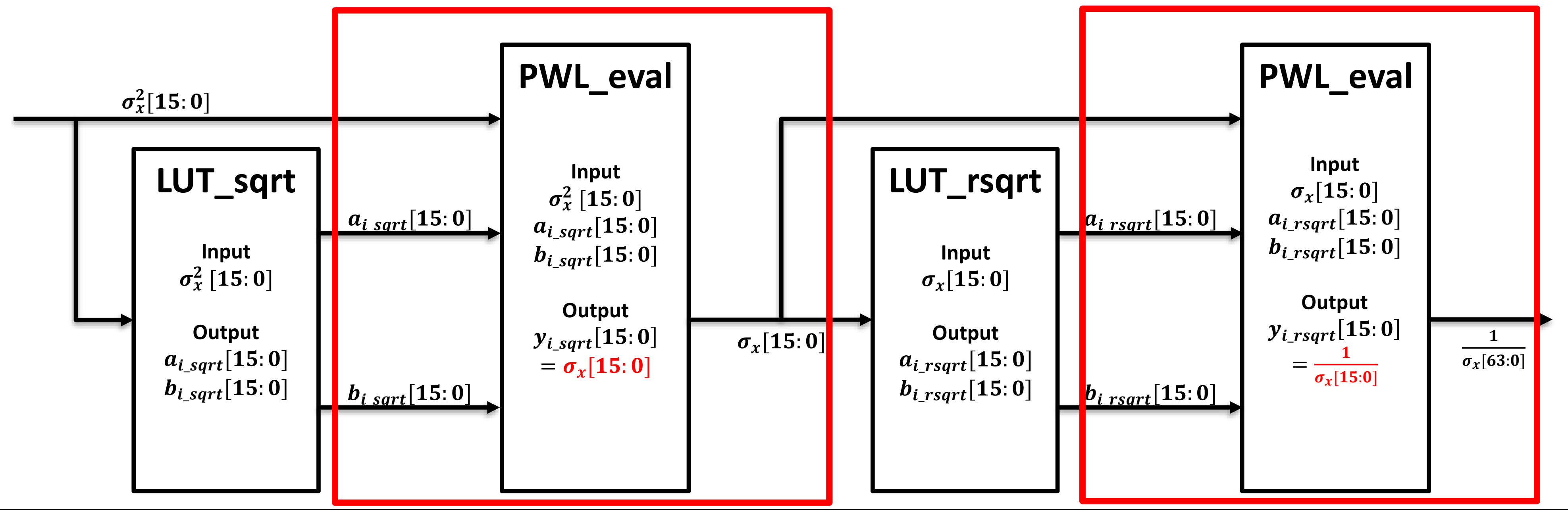
```
initial begin
    breakpoints[0] = 16'h001A; // 0.1016
    breakpoints[1] = 16'h0024; // 0.1406
    breakpoints[2] = 16'h0038; // 0.2188
    breakpoints[3] = 16'h005A; // 0.3516
    breakpoints[4] = 16'h0099; // 0.5977
    breakpoints[5] = 16'h0114; // 1.0703
    breakpoints[6] = 16'h0216; // 2.0898
    breakpoints[7] = 16'h0482; // 4.5078
    breakpoints[8] = 16'h0B45; // 11.2695

    slopes[0] = 16'hB541; // -74.7461
    slopes[1] = 16'hDFA3; // -32.3633
    slopes[2] = 16'hF339; // -12.7773
    slopes[3] = 16'hFB5F; // -4.6289
    slopes[4] = 16'hFE80; // -1.5
    slopes[5] = 16'hFF93; // -0.4258
    slopes[6] = 16'hFFE6; // -0.1016
    slopes[7] = 16'hFFFB; // -0.0195


    intercepts[0] = 16'h1164; // 17.3906
    intercepts[1] = 16'h0B78; // 11.4688
    intercepts[2] = 16'h0738; // 7.2188
    intercepts[3] = 16'h045A; // 4.3398
    intercepts[4] = 16'h027C; // 2.4844
    intercepts[5] = 16'h0154; // 1.3281
    intercepts[6] = 16'h00A6; // 0.6484
    intercepts[7] = 16'h0047; // 0.2773
end
```

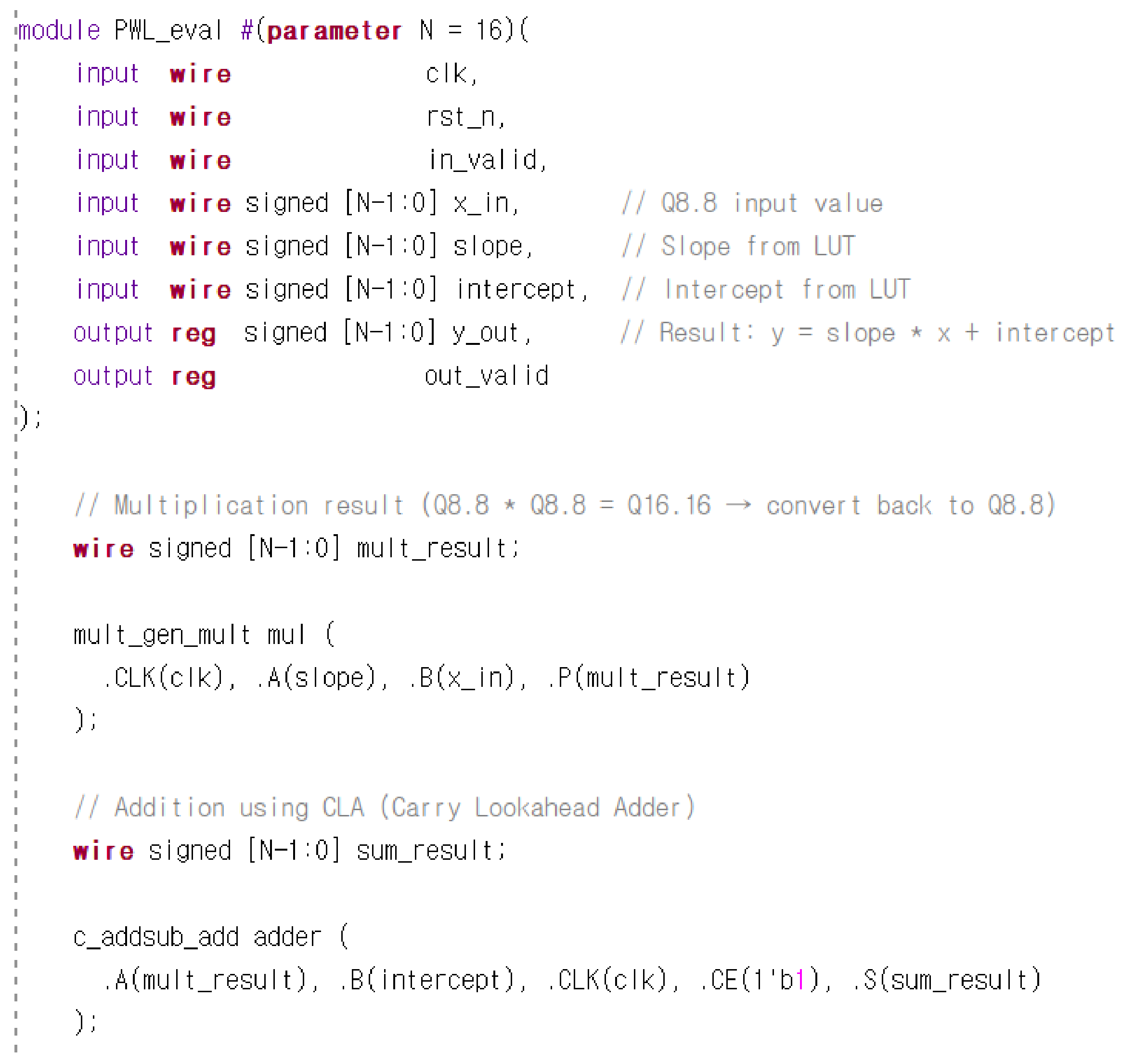
$\longrightarrow$ signed

LN Approximation Model

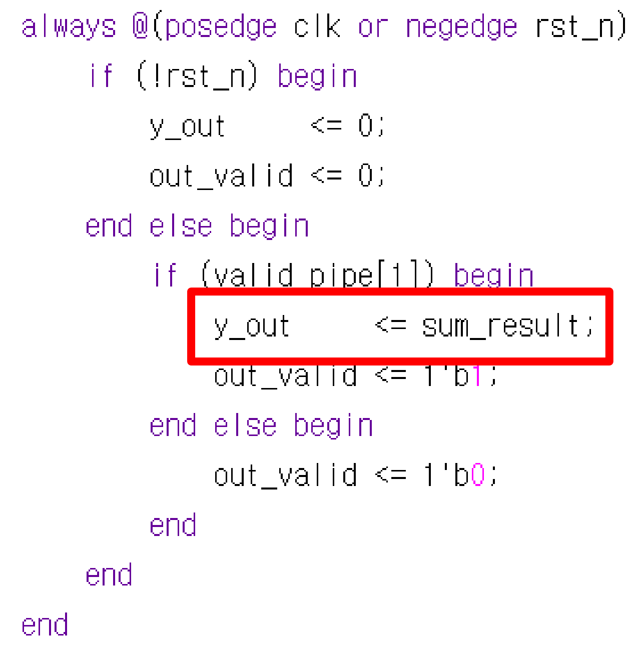


PWL_module



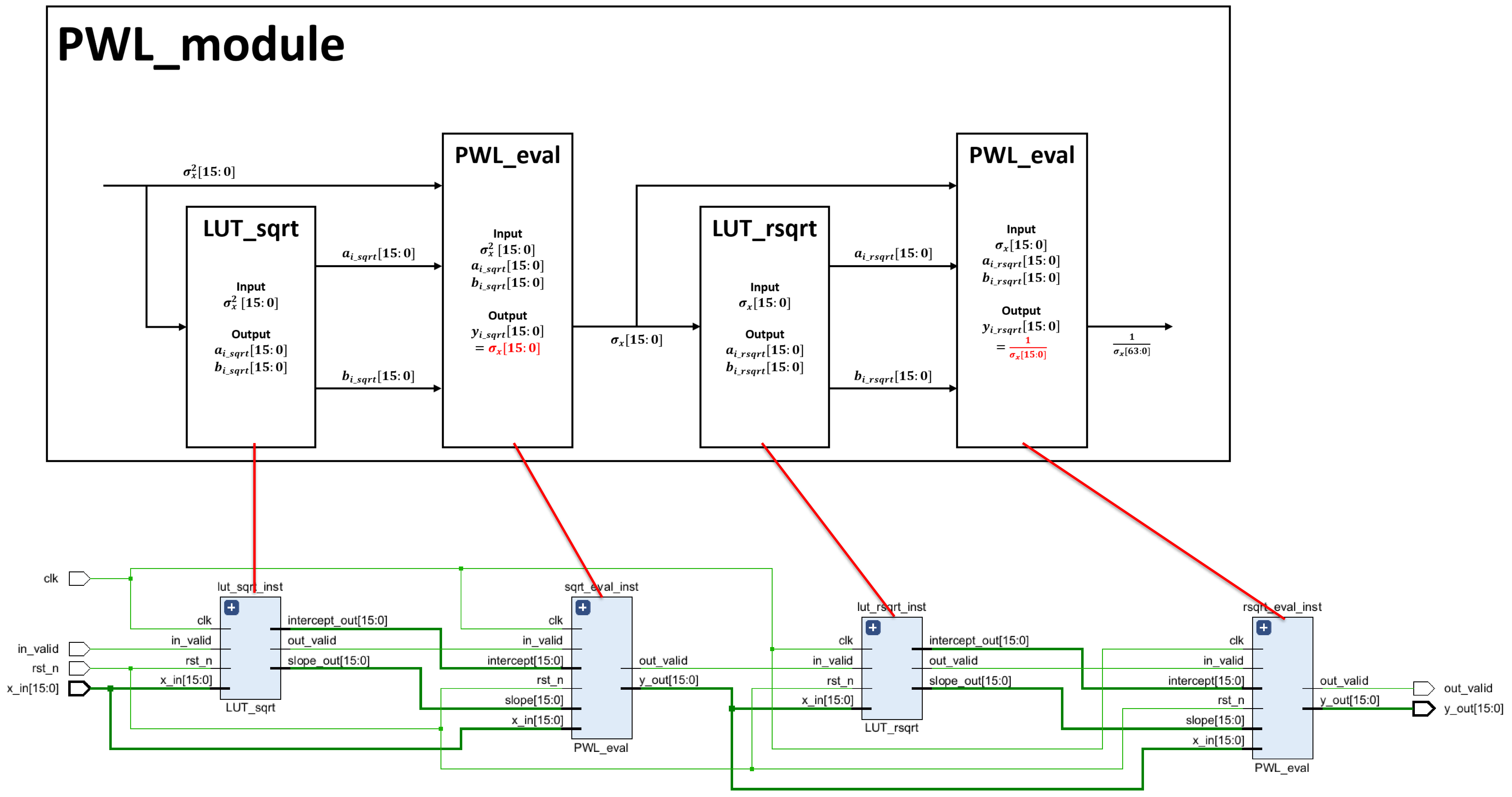


```
module PWL_eval #(parameter N = 16)(
    input  wire              clk,
    input  wire              rst_n,
    input  wire              in_valid,
    input  wire signed [N-1:0] x_in,       // Q8.8 input value
    input  wire signed [N-1:0] slope,      // Slope from LUT
    input  wire signed [N-1:0] intercept,  // Intercept from LUT
    output reg  signed [N-1:0] y_out,      // Result: y = slope * x + intercept
    output reg               out_valid
);

    // Multiplication result (Q8.8 * Q8.8 = Q16.16 → convert back to Q8.8)
    wire signed [N-1:0] mult_result;

    mult_gen_mult mul (
      .CLK(clk), .A(slope), .B(x_in), .P(mult_result)
    );

    // Addition using CLA (Carry Lookahead Adder)
    wire signed [N-1:0] sum_result;

    c_addsub_add adder (
      .A(mult_result), .B(intercept), .CLK(clk), .CE(1'b1), .S(sum_result)
    );
```

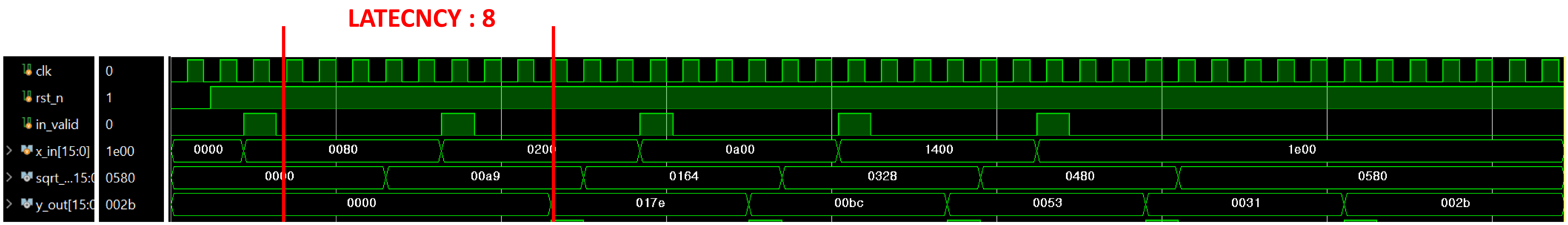


```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        y_out     <= 0;
        out_valid <= 0;
    end else begin
        if (valid_pipe[1]) begin
            y_out     <= sum_result;
            out_valid <= 1'b1;
        end else begin
            out_valid <= 1'b0;
        end
    end
end
```

LN Approximation Model

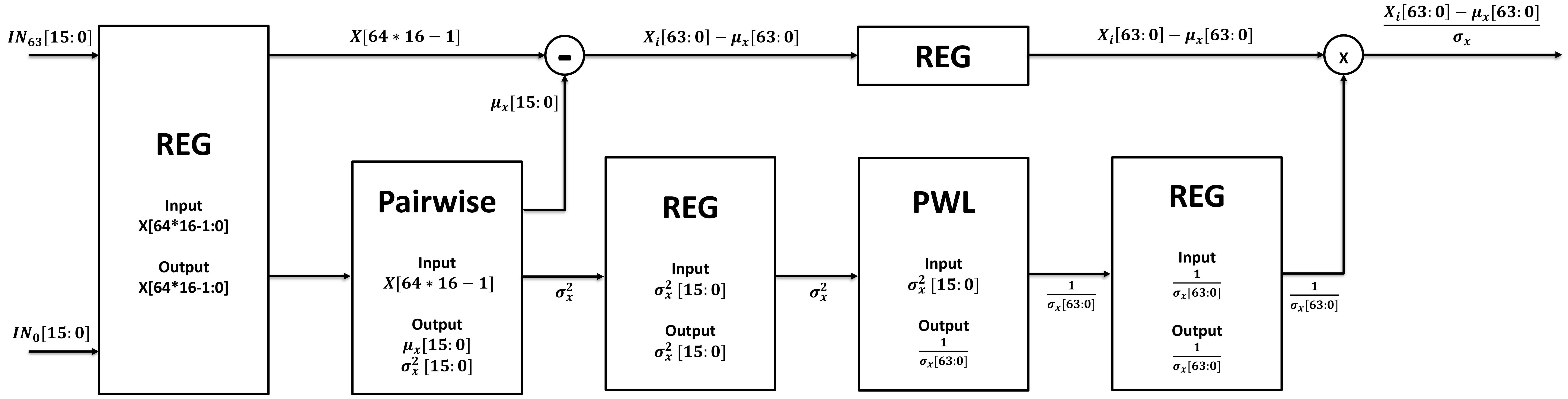


LN Approximation Model

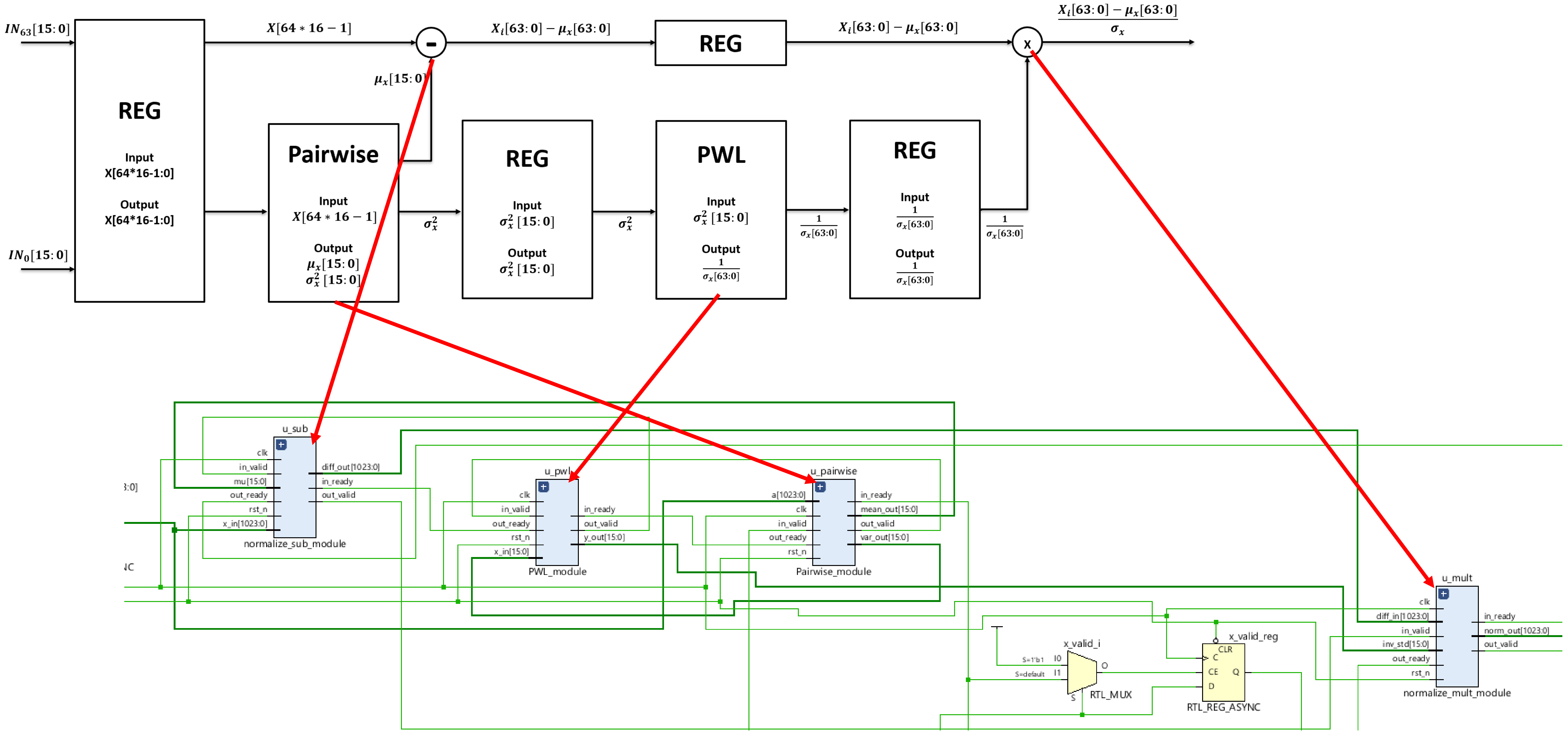

LATECNCY : 8
clk 0
rst_n 1
in_valid 0
x_in[15:0] 1e00
0000 0080 0200 0a00 1400 1e00
sqrt_...15:0 0580
0000 00a9 0164 0328 0480 0580
y_out[15:0 002b
0000 017e 00bc 0053 0031 002b

Q8.8 input hex (Dec)	X_in=0080 (0.5)	X_in=0200 (2.0)	X_in=0a00 (10.0)	X_in=1400 (20.0)	X_in=1e00 (30.0)	Accuracy
Sqrt	0x00a9 (0.660)	0x0164 (1.391)	0x0328 (3.156)	0480 (4.5)	0580 (5.5)	
Sqrt expected	0x00b5 (0.707)	0x016a (1.414)	0x032a (3.164)	0479 (4.473)	057a (5.477)	
Sqrt accuracy	93.37%	98.34%	0x99.75%	99.39%	99.57%	98.08% (error=1.92%)
reciprocal	0x017e (1.492)	0x00bc (0.734)	0x0053 (0.324)	0031 (0.191)	002b (0.168)	
reciprocal expected	0x016a (1.414)	0x00b5 (0.707)	0x0051 (0.316)	0039 (0.223)	002f (0.184)	
Reciprocal accuracy	94.48%	96.13%	97.53%	85.96%	91.49%	93.10% (error=6.90%)

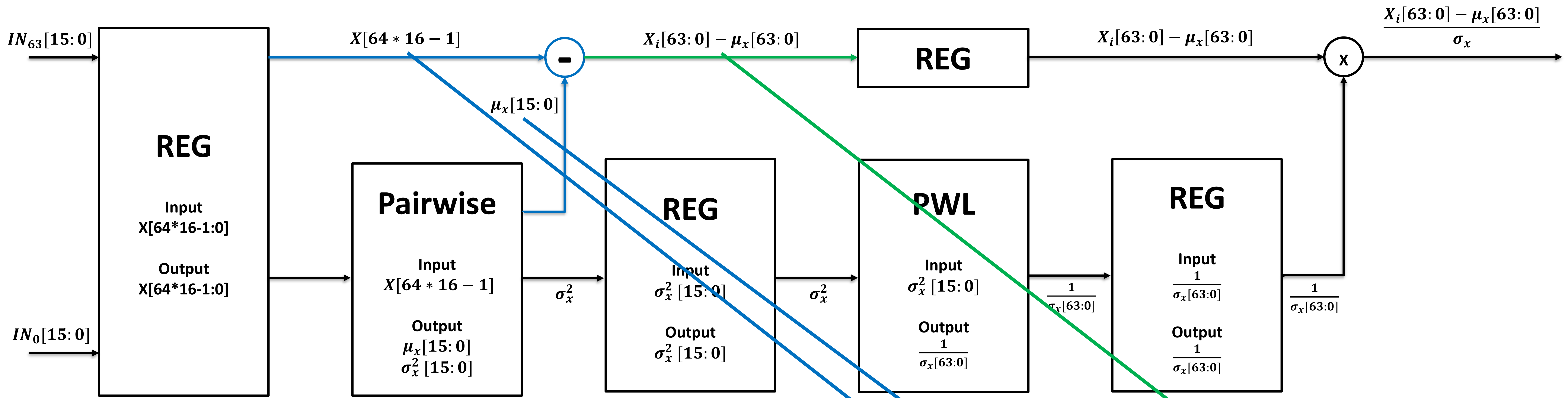
LN Approximation Model



LN Approximation Model



LN Approximation Model



```
genvar i;
generate
    for (i = 0; i < N; i = i + 1) begin : SUB_GEN
        assign x[i] = x_in[i*16 +: 16];

        c_addsub_sub sub_inst (
            .A(x[i]),
            .B(mu),
            .CLK(clk),
            .CE  (1'b1),
            .S(diff[i])
        );

        assign diff_out[i*16 +: 16] = diff[i];
    end
endgenerate
```

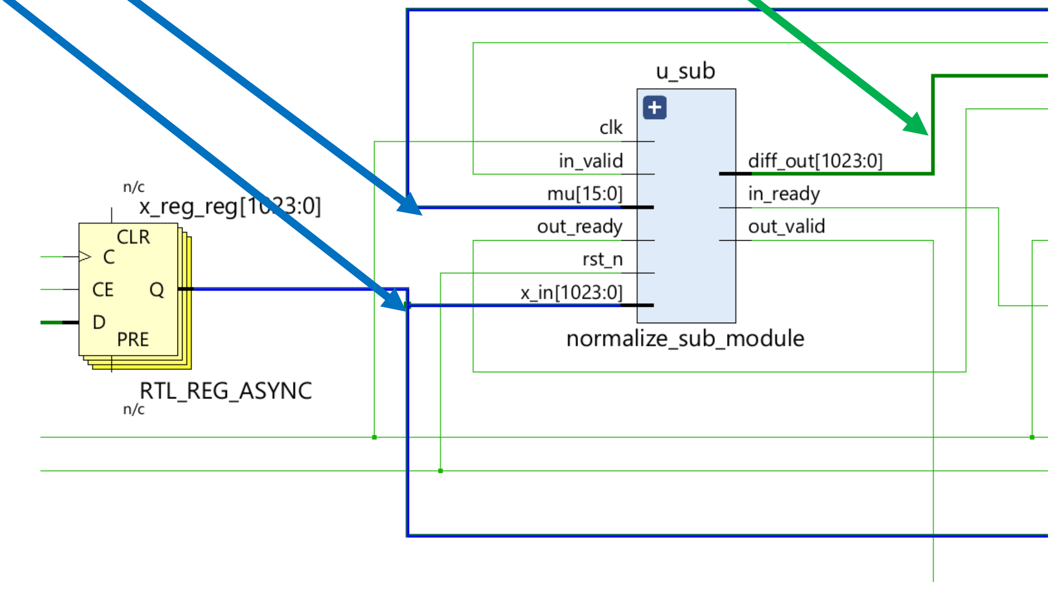
data

Input A : $X_i[63:0]$

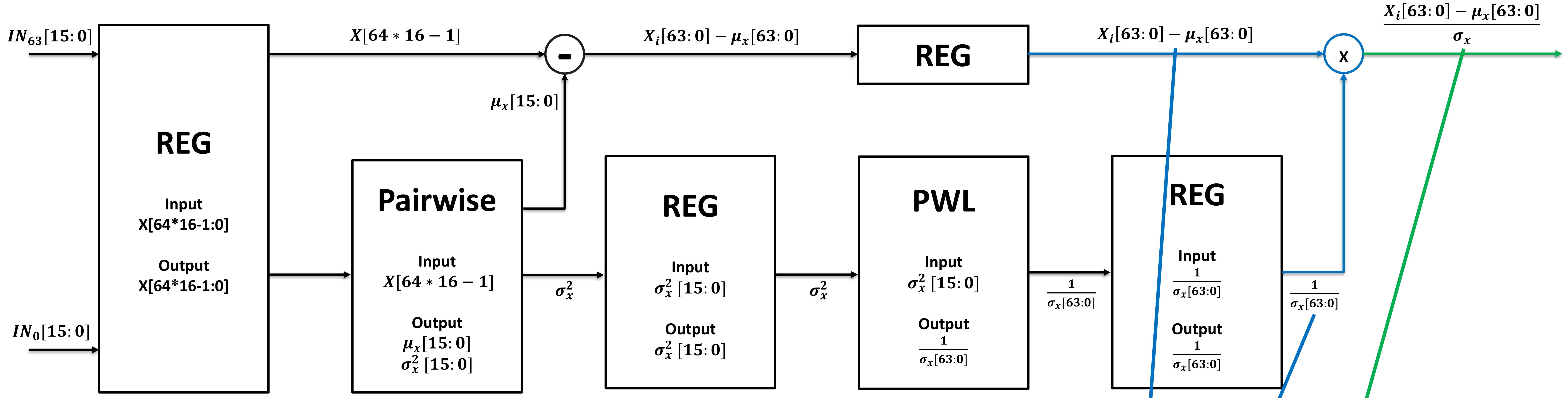
Input B : $\mu_x[63:0]$

Output : $X_i[63:0]-\mu_x[63:0]$

Using addsub IP → 1 Latency



LN Approximation Model



```
genvar i;
generate
    for (i = 0; i < N; i = i + 1) begin : MULT_GEN
        assign diff[i] = diff_in[i*16 +: 16];

        mult_gen_mult mult_inst (
            .A(diff[i]),
            .B(inv_std),
            .CLK(clk),
            .P(prod[i])
        );

        assign result[i] = prod[i][23:8];  // Q16.16 → Q8.8
        assign norm_out[i*16 +: 16] = result[i];
    end
endgenerate
```

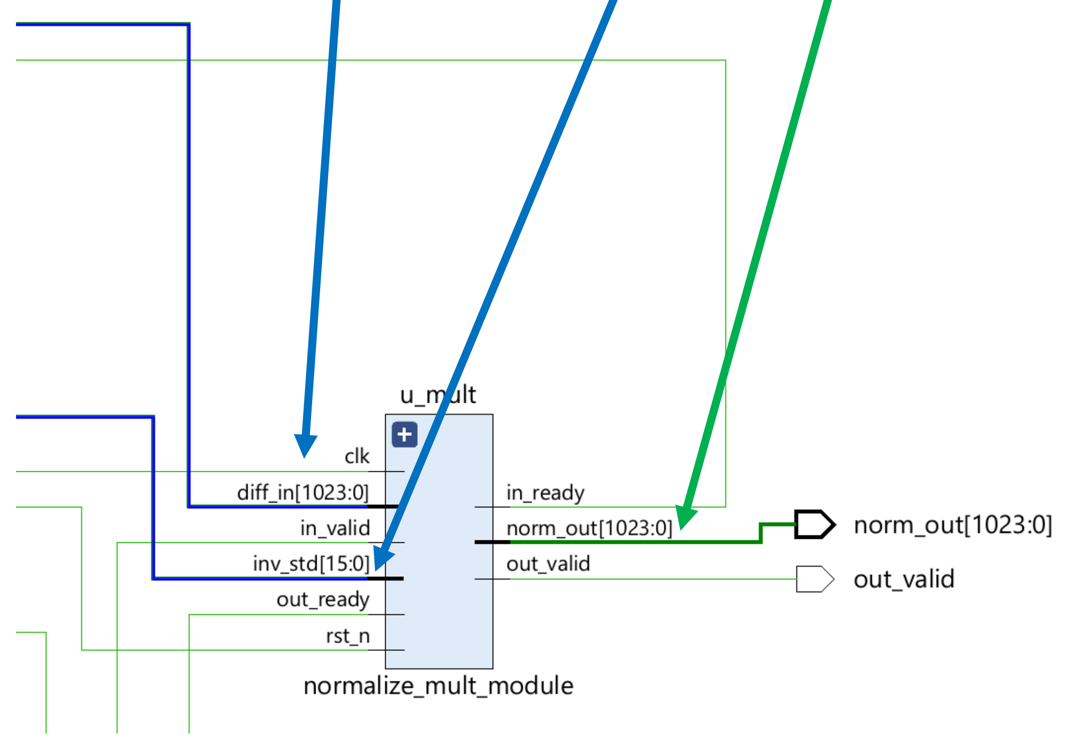
data

Input A : $X_i[63:0]-\mu_x[63:0]$

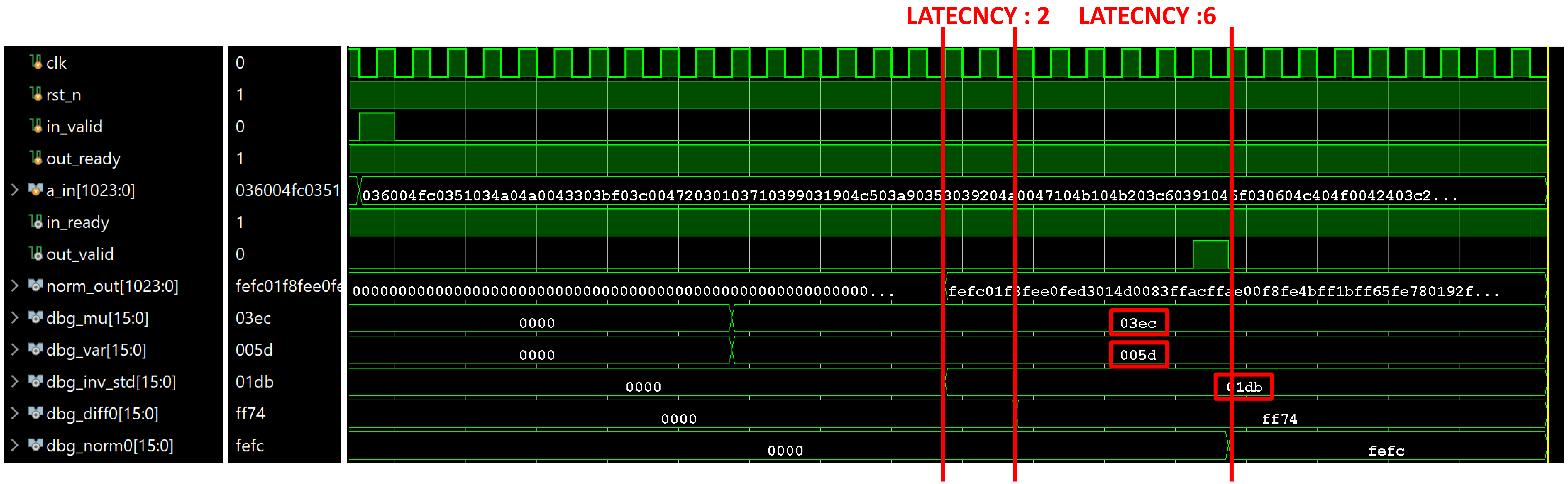
Input B : $\dfrac{1}{\sigma_x[63:0]}$

Output : $\dfrac{X_i[63:0]-\mu_x[63:0]}{\sigma_x}$

Using mult IP → 4 Latency



LN Approximation Model



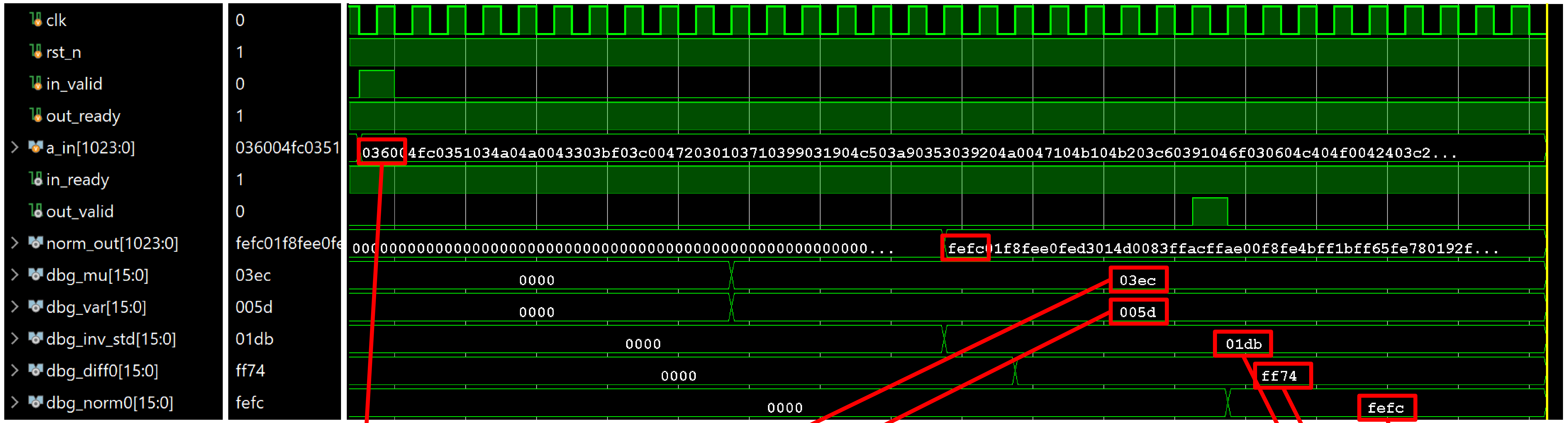
```
// 1 clock delay cycle to store reg
reg [0:0] delay_cnt;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        delay_cnt <= 0;
        out_valid <= 0;
    end else if (in_valid && in_ready) begin
        delay_cnt <= 1;
        out_valid <= 0;
    end else if (delay_cnt != 0) begin
        delay_cnt <= delay_cnt - 1;
        out_valid <= 1;  // out_valid is 1 after 1 clk delayed
    end else if (out_valid && out_ready) begin
        out_valid <= 0;
    end else begin
        out_valid <= 0;
    end
end
```

Using addsub IP → 2 Latency = 1 Latency(add) + 1 Latency(reg)
Using mult IP → 6 Latency = 4 Latency(mult) + 1 Latency(reg) + 1 Latency(dbg)

During debugging, extra 1 cycles are added for safe result verification, resulting in more delay than the actual waveform.

Thus will reduce the Latency.

LN Approximation Model

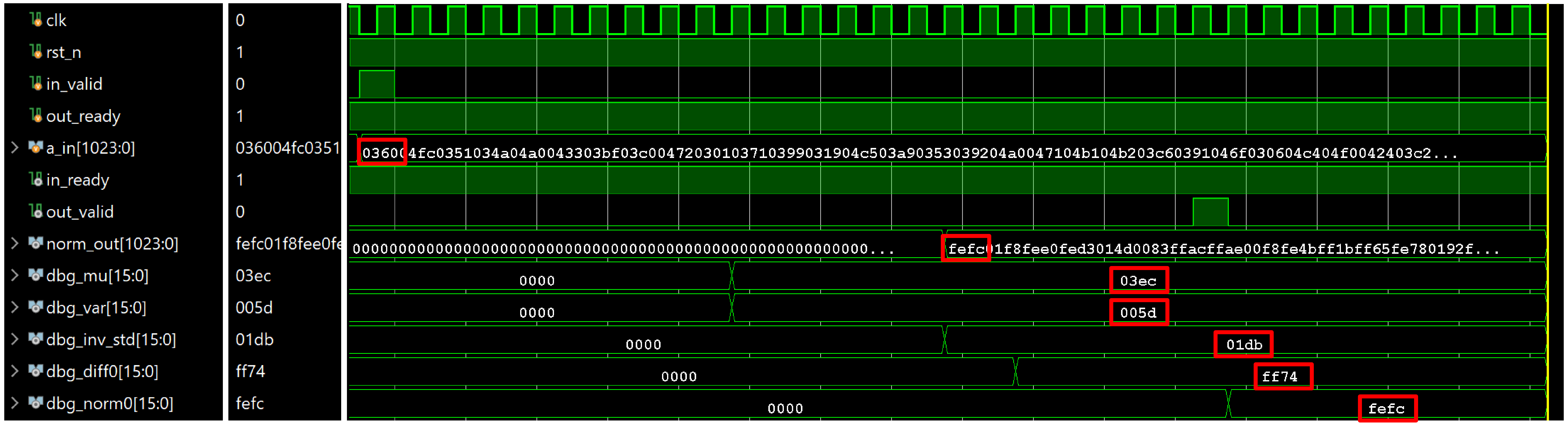


```
clk                 0
rst_n               1
in_valid            0
out_ready           1
a_in[1023:0]        036004fc0351    0360 4fc0351034a04a0043303bf03c0047203010371039903190 4c503a90353039204a0047104b104b203c60391046f030604c404f0042403c2...
in_ready            1
out_valid           0
norm_out[1023:0]    fefc01f8fee0fe  000000000000000000000000000000000000000000000000000000...   fefc01f8fee0fed3014d0083ffacffae00f8fe4bff1bff65fe780192f...
dbg_mu[15:0]        03ec            0000   03ec
dbg_var[15:0]       005d            0000   005d
dbg_inv_std[15:0]   01db            0000   01db
dbg_diff0[15:0]     ff74            0000   ff74
dbg_norm0[15:0]     fefc            0000   fefc
```

64 random input values ranging from 3.00 to 5.00

Input data	**0x0360(3.375)**				
Result_Mean	**0x03ec(3.922)**				
Result_Var	**0x005d(0.363)**	→	**0x005d(Var) * 0x00E2(a) + 0x0038(b)**	→	**$\sqrt{Var}$: 0x008A(0.539)**
$\sqrt{Var}$	**0x008A(0.539)**	→	**0x0x008A($\sqrt{Var}$) * 0xFB5F(a) + 0x045A(b)**	→	**$1/\sqrt{Var}$: 0x01DB(1.855)**
Result_Diff	**0x0360(3.375) / 0x03ec(3.922)**	→	**0x0360(3.375) - 0x03ec(3.922)**	→	**Diff : 0xFF74(-0.547)**
Result_Norm	**0xFF74(-0.547) / 0x01DB(1.855)**	→	**0xFF74(-0.547) * 0x01DB(1.855)**	→	**Norm : 0xFEFC(-1.016)**

LN Approximation Model



```
clk                 0
rst_n               1
in_valid            0
out_ready           1
a_in[1023:0]        036004fc0351     03600 4fc0351034a04a0043303bf03c00472030103710399031904c503a90353039204a0047104b104b203c60391046f030604c404f0042403c2...
in_ready            1
out_valid           0
norm_out[1023:0]    fefc01f8fee0fe   0000000000000000000000000000000000000000000000000000000...   fefc01f8fee0fed3014d0083ffacffae00f8fe4bff1bff65fe780192f...
dbg_mu[15:0]        03ec             0000   03ec
dbg_var[15:0]       005d             0000   005d
dbg_inv_std[15:0]   01db             0000   01db
dbg_diff0[15:0]     ff74             0000   ff74
dbg_norm0[15:0]     fefc             0000   fefc
```

64 random input values ranging from 3.00 to 5.00

	Real	Approx	Accuracy(%)
Input Value	3.37	0x0360(3.375)	99.85
Mean	3.937	0x03ec(3.922)	99.61
Variance	0.368	0x005d(0.363)	98.64
1/Variance	1.649	0x01db(1.855)	87.51
Result(Norm)	**-0.935**	**0xfefc(-1.016)**	**91.34**
Norm mean	0	0.004	-

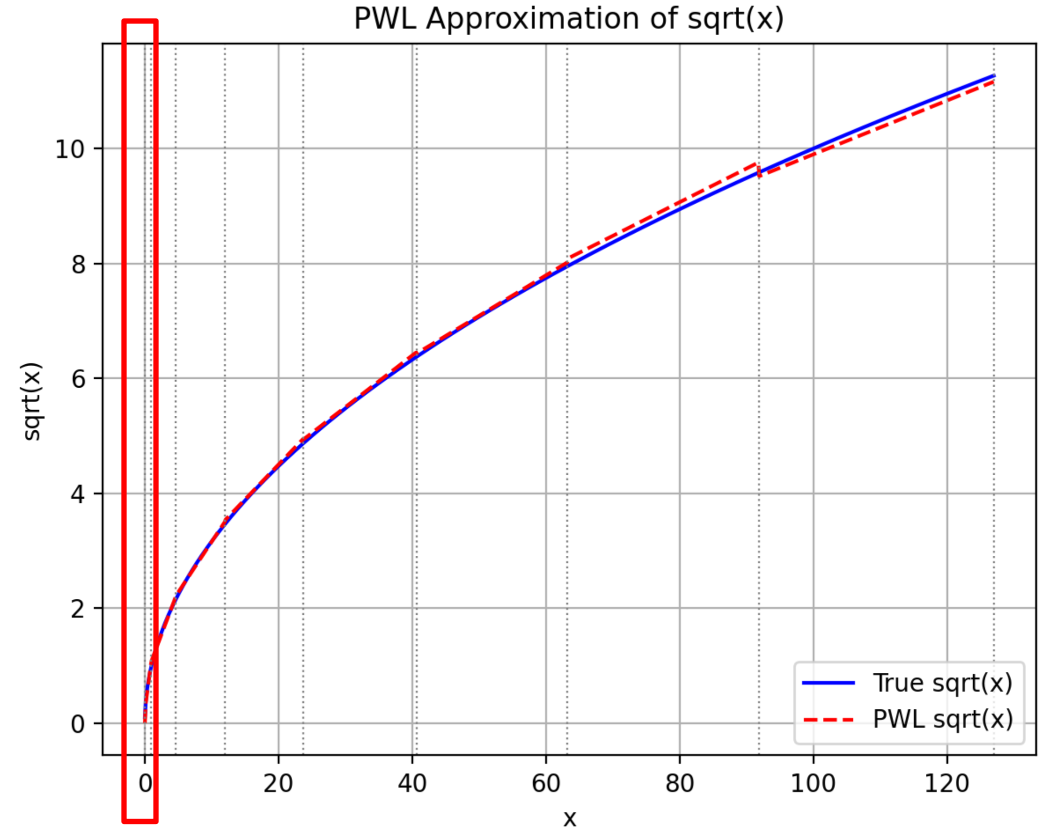
Accuracy(%)	data count
90 ~ 100	31
80 ~ 90	26
Other	7

LN Approximation Model

64 random input values ranging from 3.00 to 5.00

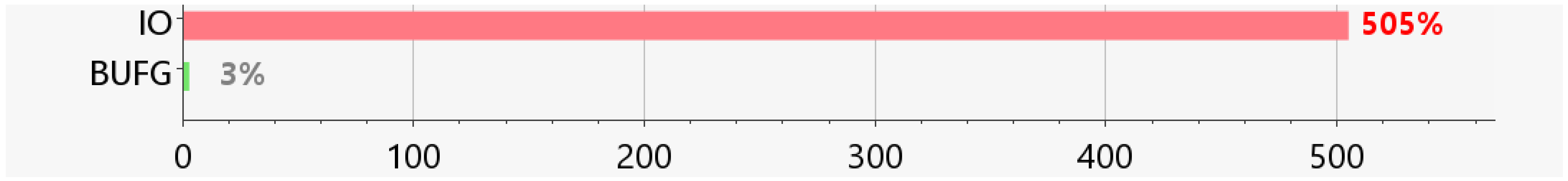
Accuracy dropped due to the narrow **input range (var ≈ 0)**,
which is much **smaller than** the **expected real-case variance (20 ~ 30).**

Therefore, **LUT range will be redefined** based on the **actual input distribution.**



Apply FPGA

Input : 64 * 16bit → [1023:0] : 1024EA
Output 64 * 16bit → [1023:0] : 1024EA
Other port(clk, rst…) : 6EA

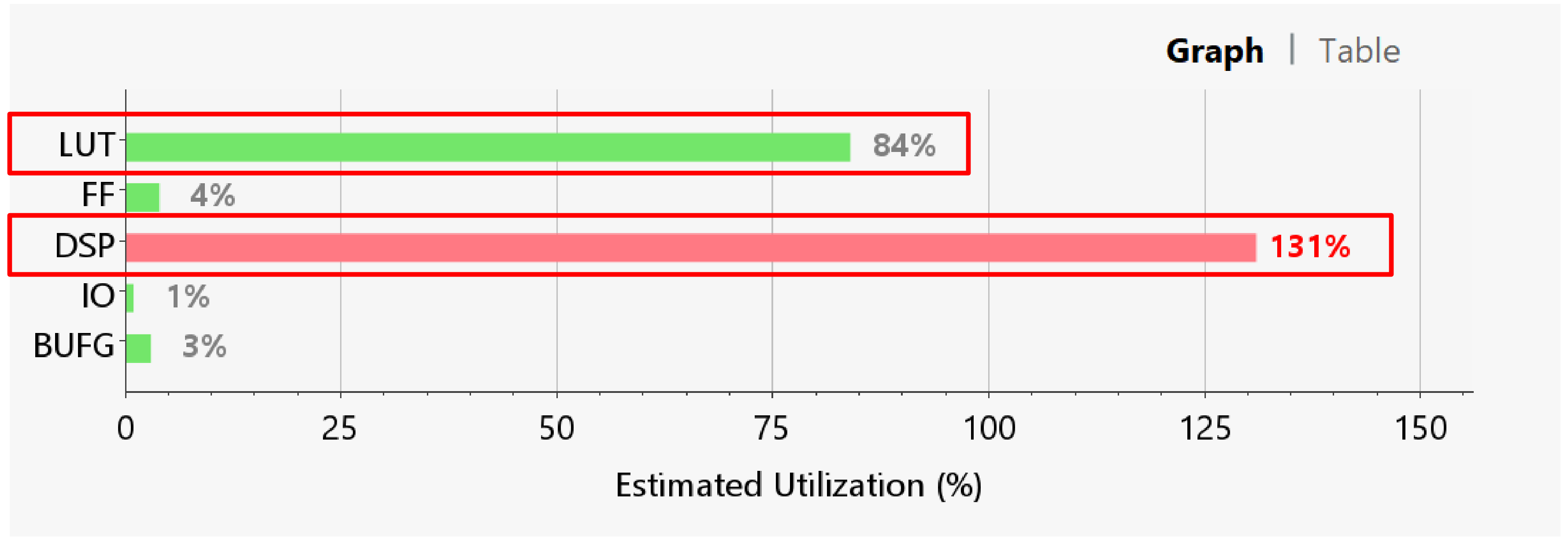


Need 2054 I/O port, But only exist 210 I/O port
→ Using FSM(Finite State Machine)

Apply FPGA

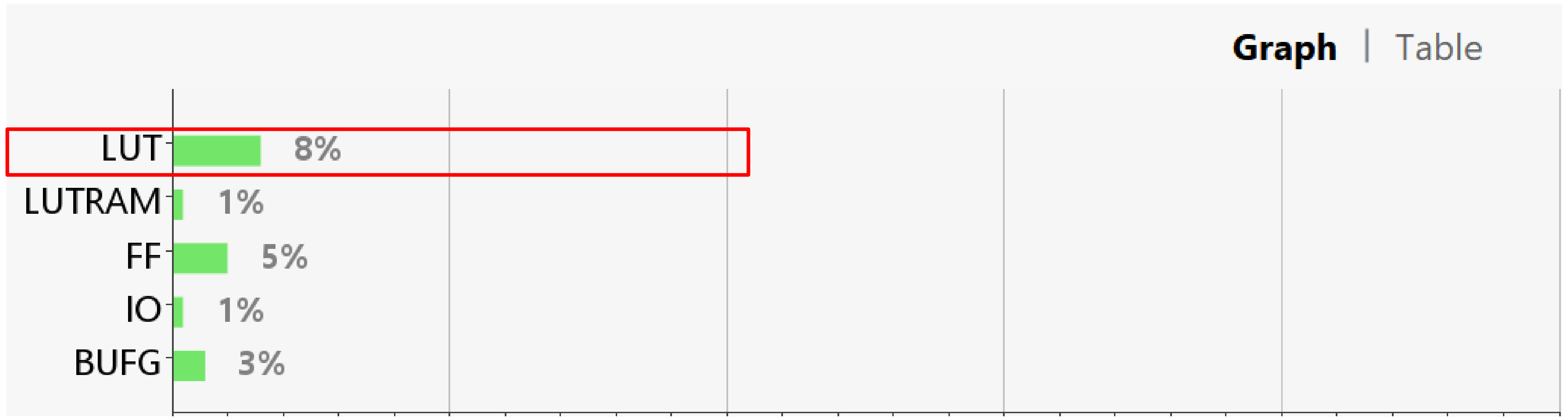
When Calculate add, sub, mult.

When not using IP



Too many used LUT, DSP → Consider IP

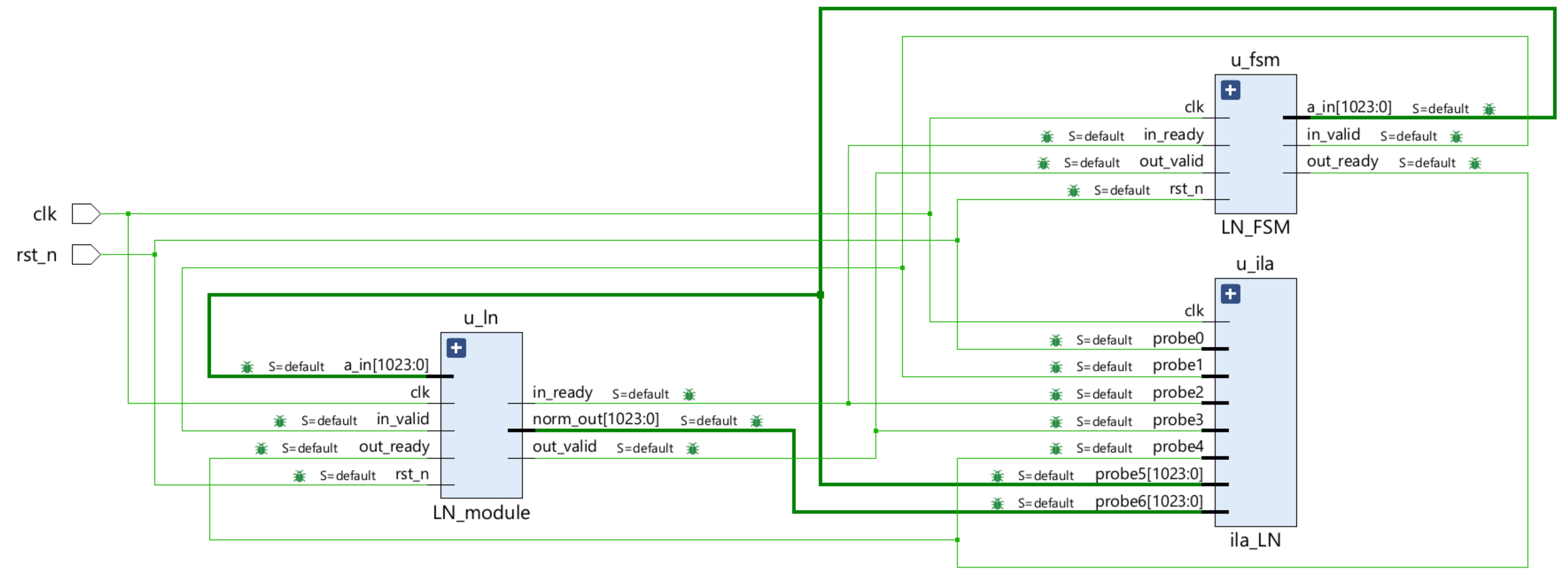
Using IP



Apply FPGA

Logic Overview

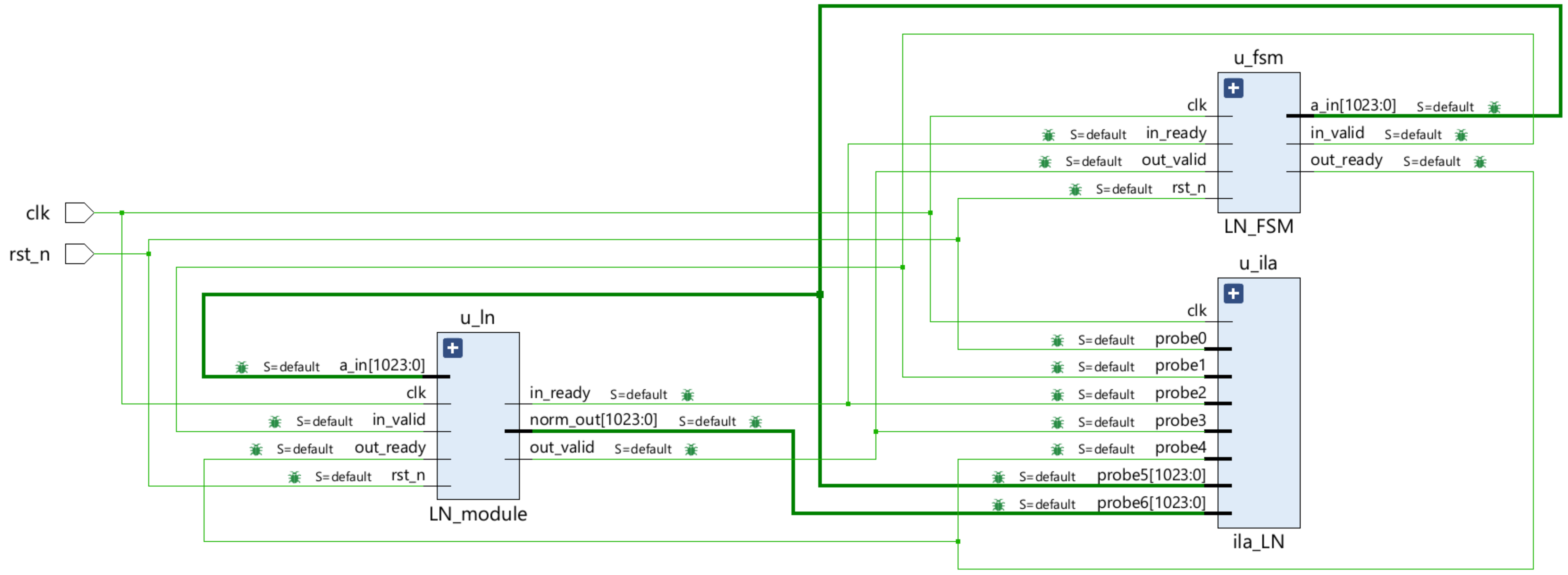
Top_module	: include LN_module, FSM, ILA.
LN_module	: Performs normalization on input data
FSM	: Used to verify correct operation on FPGA before **applying BRAM**
ILA	: Used to observe signal values and **confirm** correct **operation** on **FPGA**



Apply FPGA

Logic Overview

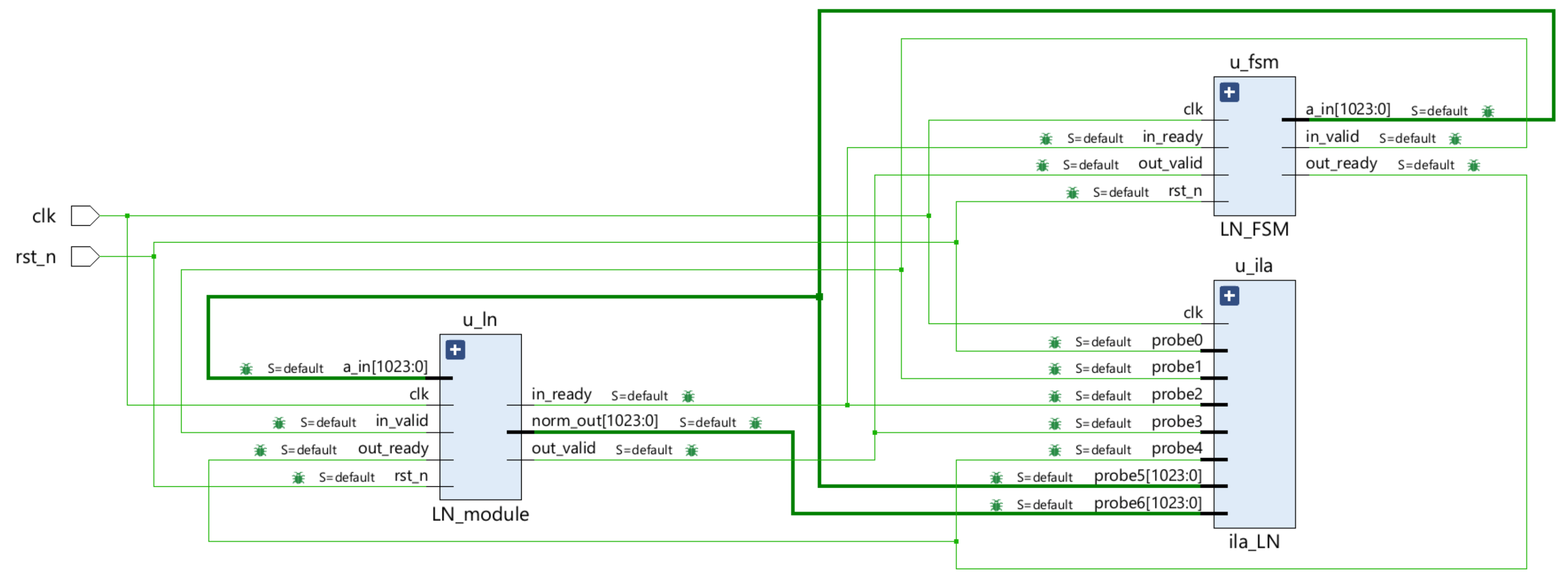
```
LN_FSM u_fsm (
.clk(clk),              // 1bit, input
.rst_n(rst_n),          // 1bit, input
.in_valid(in_valid),    // 1bit, input
.in_ready(in_ready),    // 1bit, input
.a_in(a_in),            // 64*16bit => 1024, output
.out_valid(out_valid),  // 1bit, output
.out_ready(out_ready)   // 1bit, output
```



```
LN_module u_ln (
    .clk(clk),                // 1bit, input
    .rst_n(rst_n),            // 1bit, input
    .in_valid(in_valid),      // 1bit, input
    .in_ready(in_ready),      // 1bit, output
    .out_valid(out_valid),    // 1bit, output
    .out_ready(out_ready),    // 1bit, input
    .a_in(a_in),              // 64*16bit => 1024, input
    .norm_out(norm_out)       // 64*16bit => 1024, output
);
```

```
ila_LN u_ila (
    .clk(clk),             // 1bit, input
    .probe0(rst_n),        // 1bit, input
    .probe1(in_valid),     // 1bit, input
    .probe2(in_ready),     // 1bit, input
    .probe3(out_valid),    // 1bit, input
    .probe4(out_ready),    // 1bit, input
    .probe5(a_in),         // 64*16bit => 1024, input
    .probe6(norm_out)      // 64*16bit => 1024, input
);
```

Apply FPGA



However, Error occurred due to excessive resource usage during the pairwise computation of the 64 input values.

Therefore, the data input will be changed to 32 elements, as proposed in the paper.

Apply FPGA

Error about 64 Inputs.

Error due to **excessive resource usage.**

Occurred during pairwise computation **(64 inputs).**

Plan to Solve : change 32 Inputs

Change input size : 64 → 32 elements

Mult IP : 1-cycle → 4-cycle (allows higher freq.)

Latency Change

Pairwise stage : 6 cycles → 15 cycles

GPT-2 model(768) : +12 cycles

Removed 64→32 conversion stage : −15 cycles

Conclusion

Higher frequency possible

Net latency **reduced by 3 cycles**

Reduces resource consumption compared to 64-input design

Plans for Next

- Change the number of input data from 64 to 32.
- Redefine the LUT based on the updated input data.
- Reduce unnecessary latency (originally added for debugging purposes).
- Enable continuous data input through pipelining. +) also using BRAM
- Upload the design to the FPGA and compare it with the baseline model.