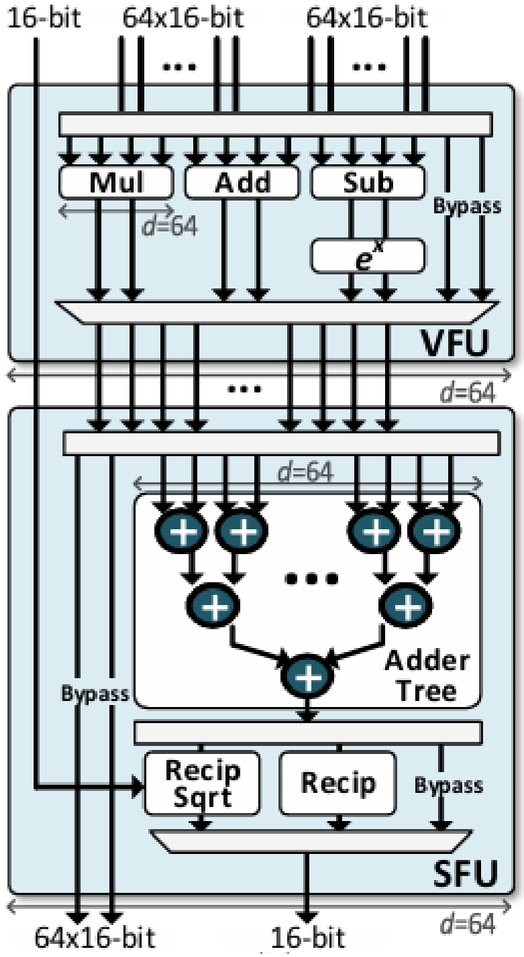
LayerNorm FPGA
(Standard & Approximation)

2025.7.15.

Sangyun Kim
Eunju Kim

CONTENTS

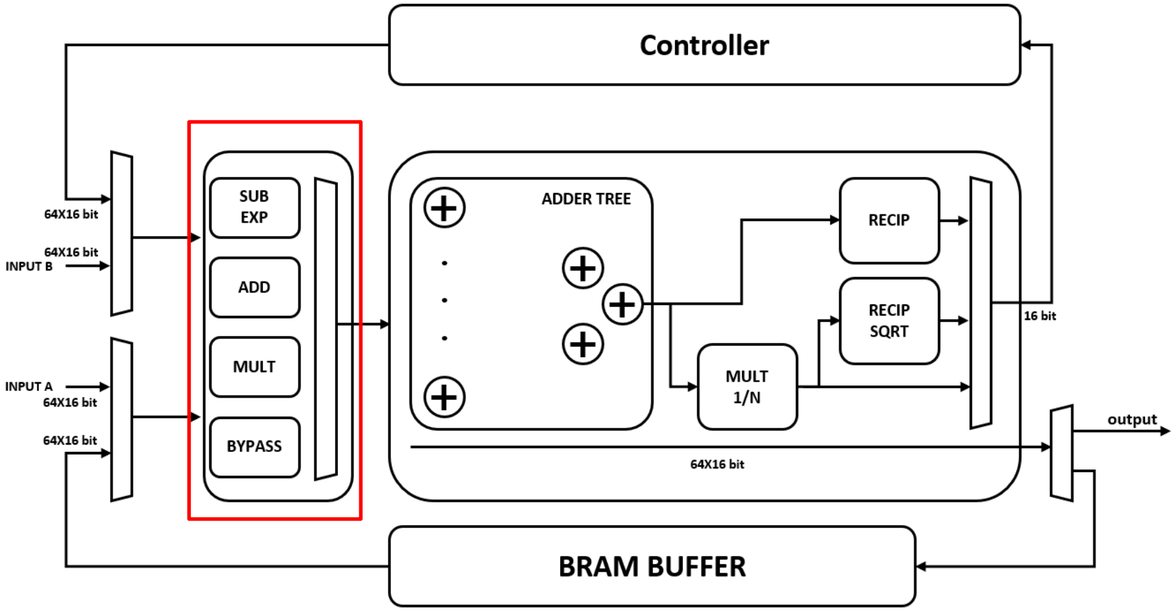
Modules Required for LayerNorm Acceleration

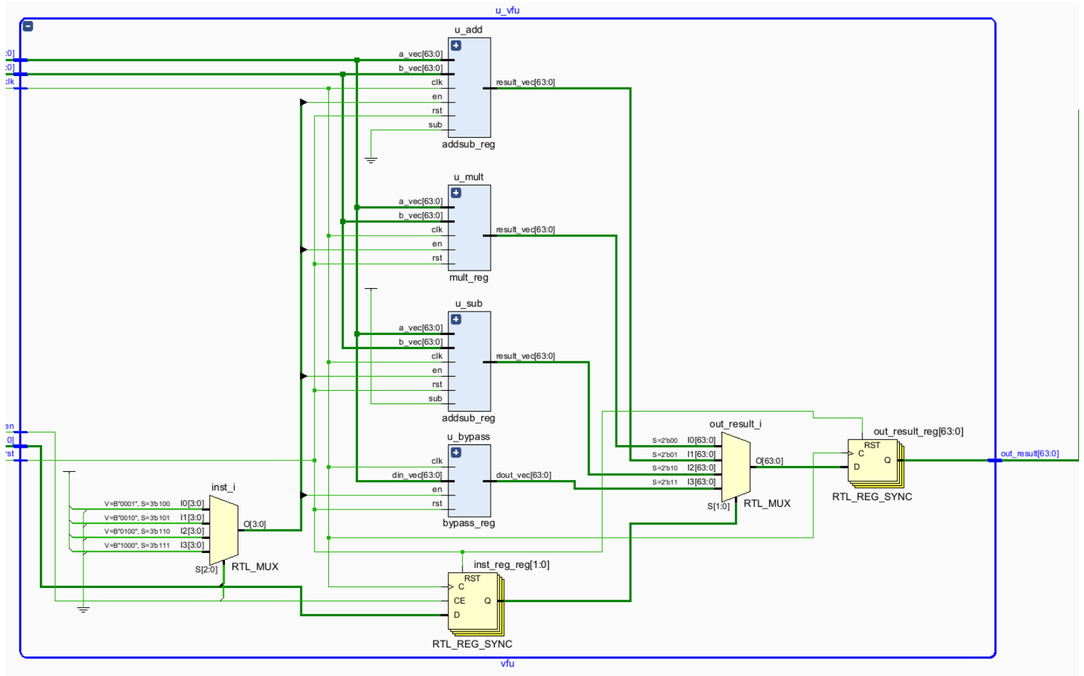


Vector Processing Unit

Module	Used in Equation	module
Adder(Subtractor)	$x_i - \mu, etc.$	IP Catalog
Multiplier	$(x_i - \mu)^2, etc.$	IP Catalog
Accummulator(Adder Tree)	$\sum \boldsymbol{x_i}$	IP Catalog(Adder)
Reciprocal(Devide)	$\mu = \frac{1}{N}\sum x_i$	IP Catalog
Reciprocal Sqrt	$\frac{x_i - \mu}{\sqrt{\sigma_i^2 + \epsilon}}$	IP Catalog

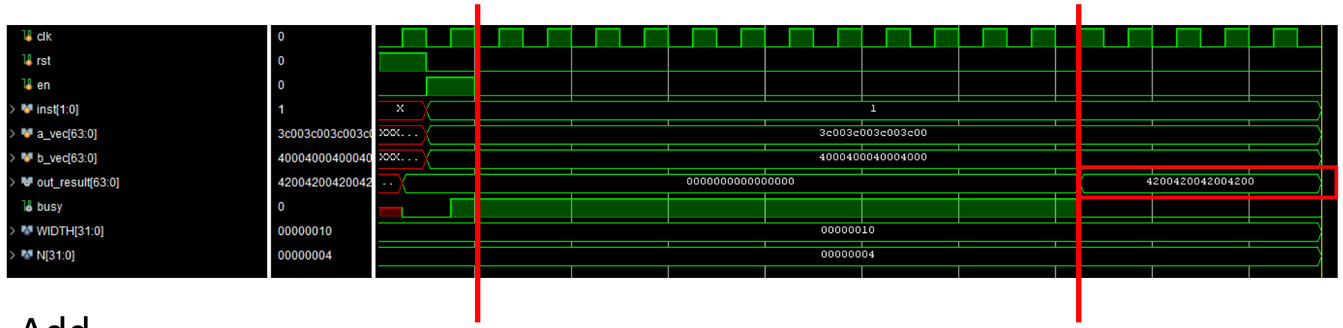
Schematic of VFU





Check the IP Catalog Latency

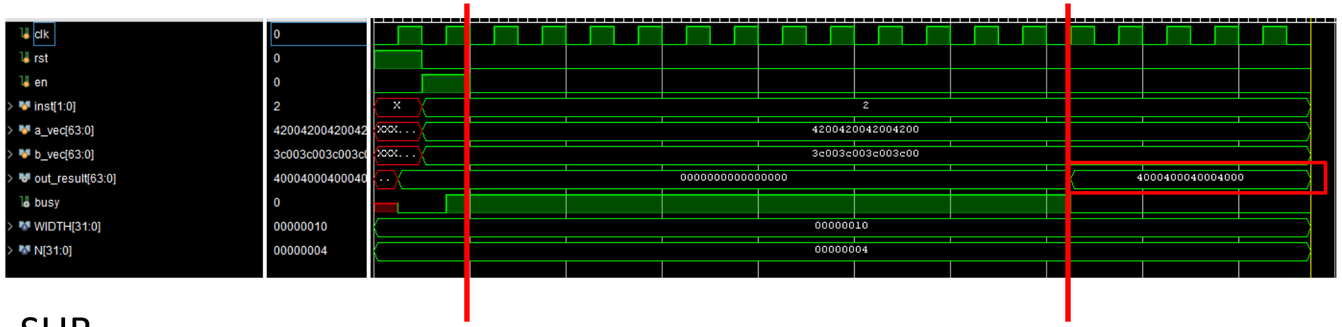
Module	IP	latency
ADD(SUB)	**Using floating_point_(add)**	**12**
ADDER TREE	Using floating_point_(add)	12 * 6 (6 Level : 64 → 32 .. →1)
MULT	Using floating_point_(mult)	7
MULT 1/N	floating_point_(div)	16
RECIP_SQRT	floating_point_(rsqrt)	5



Add

Latency : 12

ADD	
Input A	3c00(1)
Input B	4000(2)
Result	**4200(3)**



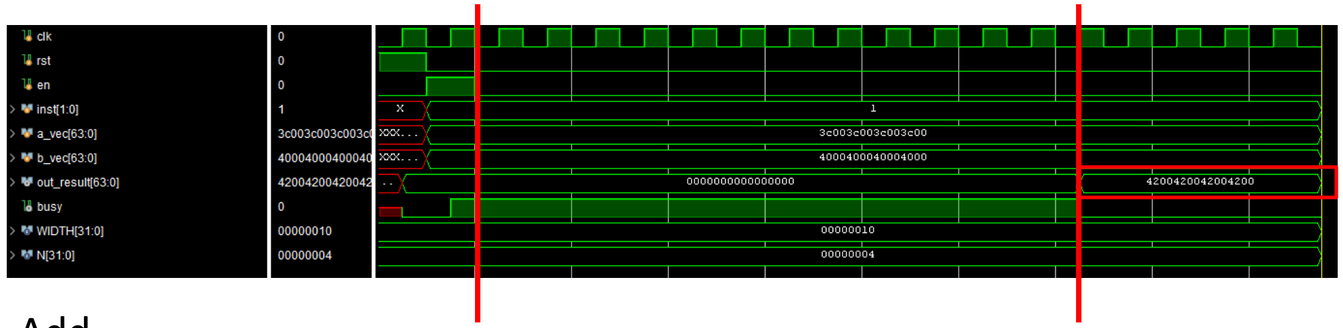
SUB

SUB	
Input A	4200(3)
Input B	3c00(1)
Result	**4000(2)**

Apply CLA the reduce the Latency and Q8.8(8 int, 8 fraction)

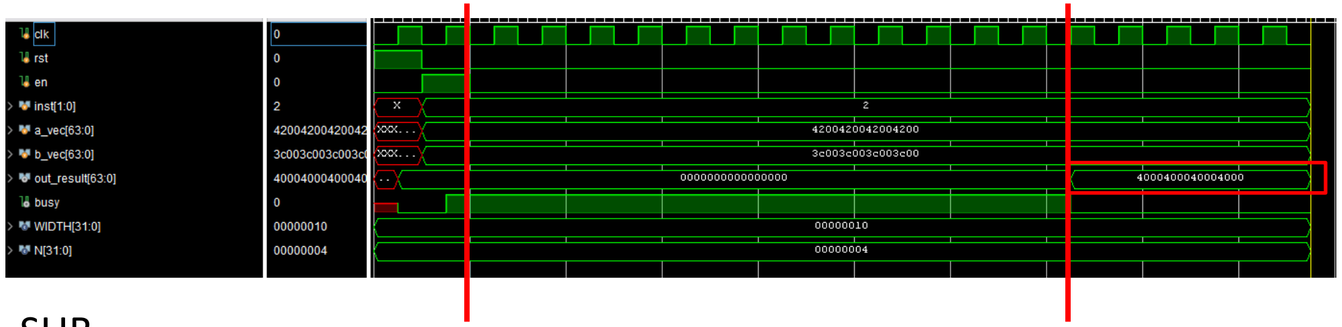
Check the IP Catalog Latency

Module	IP	latency
ADD(SUB)	**Using floating_point_(add)**	**12**
ADDER TREE	Using floating_point_(add)	12 * 6 (6 Level : 64 → 32 .. →1)
MULT	Using floating_point_(mult)	7
MULT 1/N	floating_point_(div)	16
RECIP_SQRT	floating_point_(rsqrt)	5



Add

Latency : 12



SUB

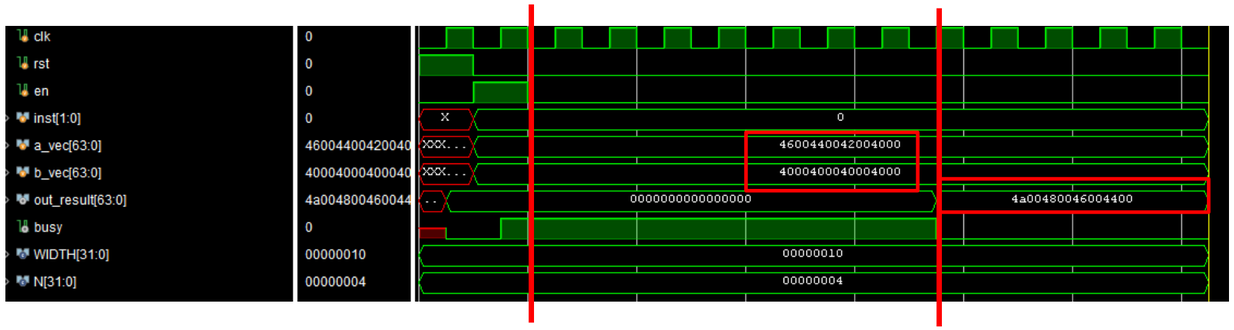
```
module addsub_reg #(
    parameter N = 4,
    parameter WIDTH = 16
)(
    input  wire clk,
    input  wire rst,
    input  wire en,
    input  wire sub,  // 0: add, 1: sub
    input  wire [N*WIDTH-1:0] a_vec,
    input  wire [N*WIDTH-1:0] b_vec,
    output wire [N*WIDTH-1:0] result_vec
);

genvar i;
generate
    for (i = 0; i < N; i = i + 1) begin : addsub_array
        wire [1:0] op_i;
        wire       en_i;
        assign op_i = {1'b0, sub};
        assign en_i = en;

        floating_point_addsub u_fp_addsub (
            .aclk(clk),
            .s_axis_a_tvalid(en_i),
            .s_axis_a_tdata(a_vec[i*WIDTH +: WIDTH]),
            .s_axis_b_tvalid(en_i),
            .s_axis_b_tdata(b_vec[i*WIDTH +: WIDTH]),
            .s_axis_operation_tvalid(en_i),
            .s_axis_operation_tdata(op_i),
            .m_axis_result_tvalid(),
            .m_axis_result_tready(1'b1),
            .m_axis_result_tdata(result_vec[i*WIDTH +: WIDTH])
        );
    end
endgenerate

endmodule
```

IP catalog

Apply CLA the reduce the Latency and Q8.8(8 int, 8 fraction)

Check the IP Catalog Latency

Module	IP	latency
ADD(SUB)	Using floating_point_(add)	12
ADDER TREE	Using floating_point_(add)	12 * 6 (6 Level : 64 → 32 .. →1)
MULT	**Using floating_point_(mult)**	**7**
MULT 1/N	floating_point_(div)	16
RECIP_SQRT	floating_point_(rsqrt)	5



Mult

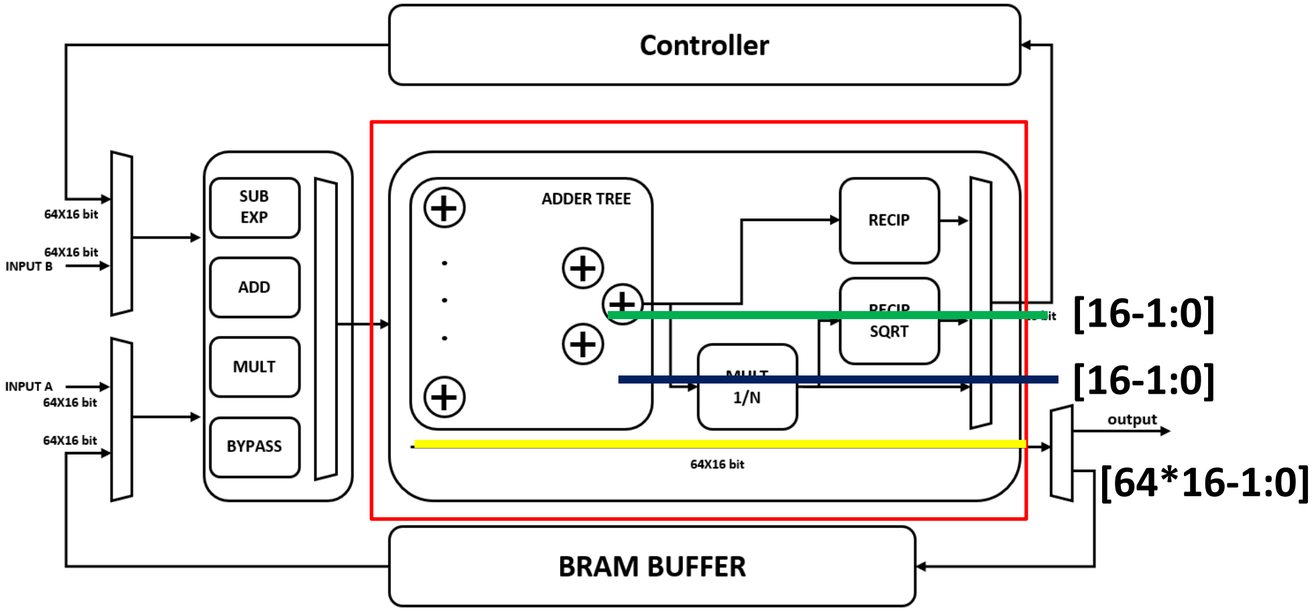
Latency : 7

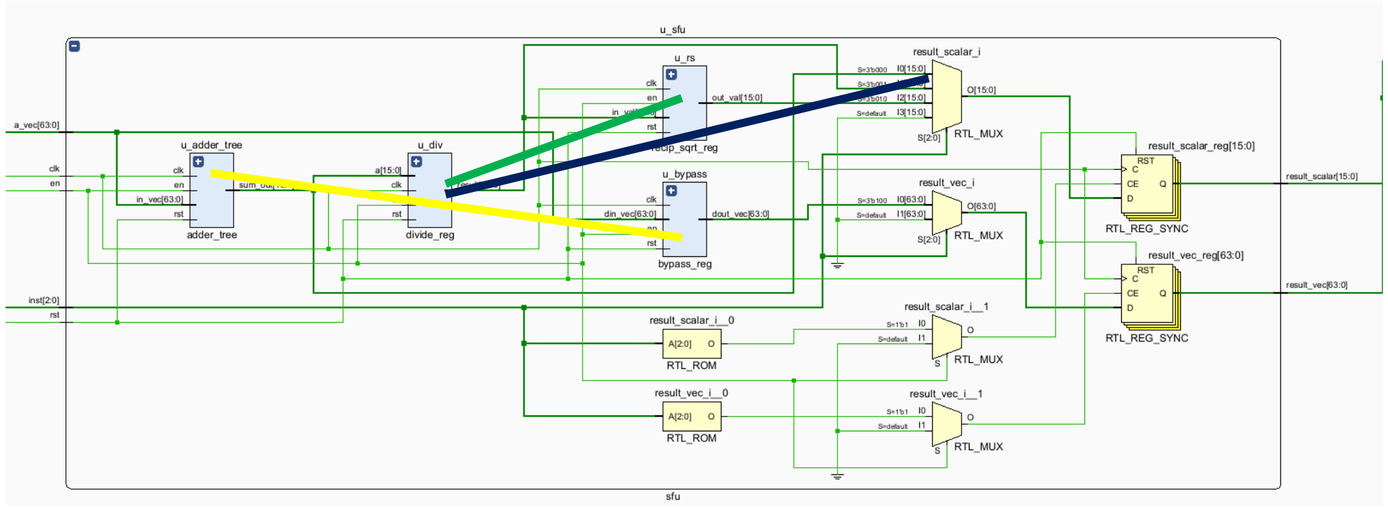
MULT				
Input A	4600(6)	4400(4)	4200(3)	4000(2)
Input B	4000(2)	4000(2)	4000(2)	4000(2)
Result	4a00(12)	4800(8)	4600(6)	4400(4)

```
module mult_reg #(
    parameter N = 4,
    parameter WIDTH = 16
)(
    input  wire clk,
    input  wire rst,
    input  wire en,
    input  wire [N*WIDTH-1:0] a_vec,
    input  wire [N*WIDTH-1:0] b_vec,
    output wire [N*WIDTH-1:0] result_vec
);

genvar i;
generate
    for (i = 0; i < N; i = i + 1) begin : mult_array
        wire en_i;
        assign en_i = en;

        floating_point_mult u_fp_mult (
            .aclk(clk),
            .s_axis_a_tvalid(en_i),
            .s_axis_a_tdata(a_vec[i*WIDTH +: WIDTH]),
            .s_axis_b_tvalid(en_i),
            .s_axis_b_tdata(b_vec[i*WIDTH +: WIDTH]),
            .m_axis_result_tvalid(),
            .m_axis_result_tready(1'b1),
            .m_axis_result_tdata(result_vec[i*WIDTH +: WIDTH])
        );
    end
endgenerate

endmodule
```

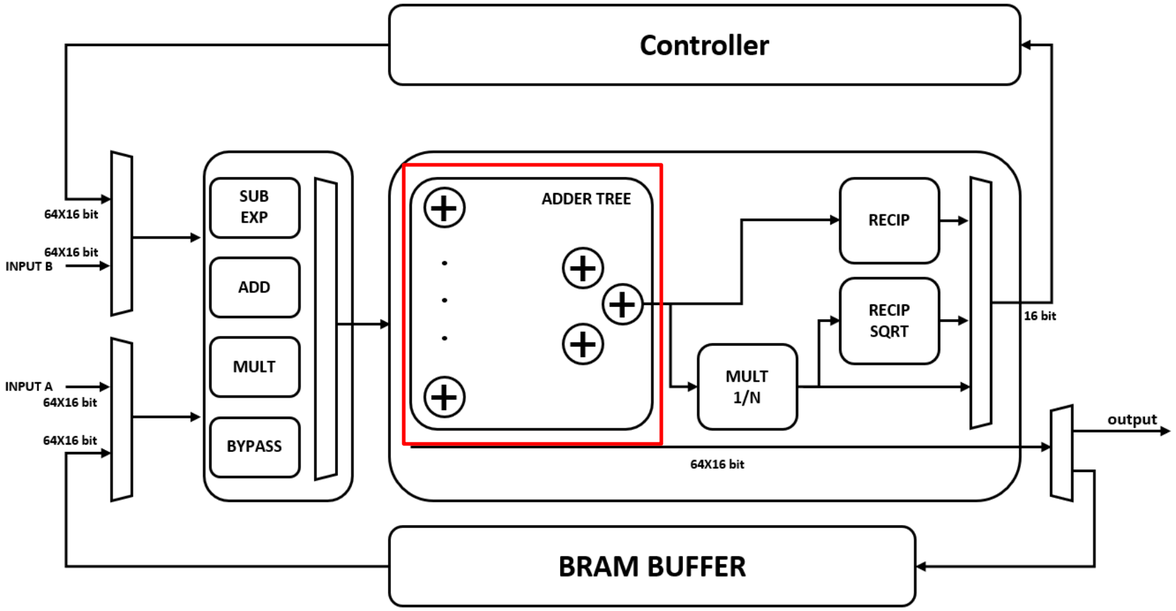
IP catalog

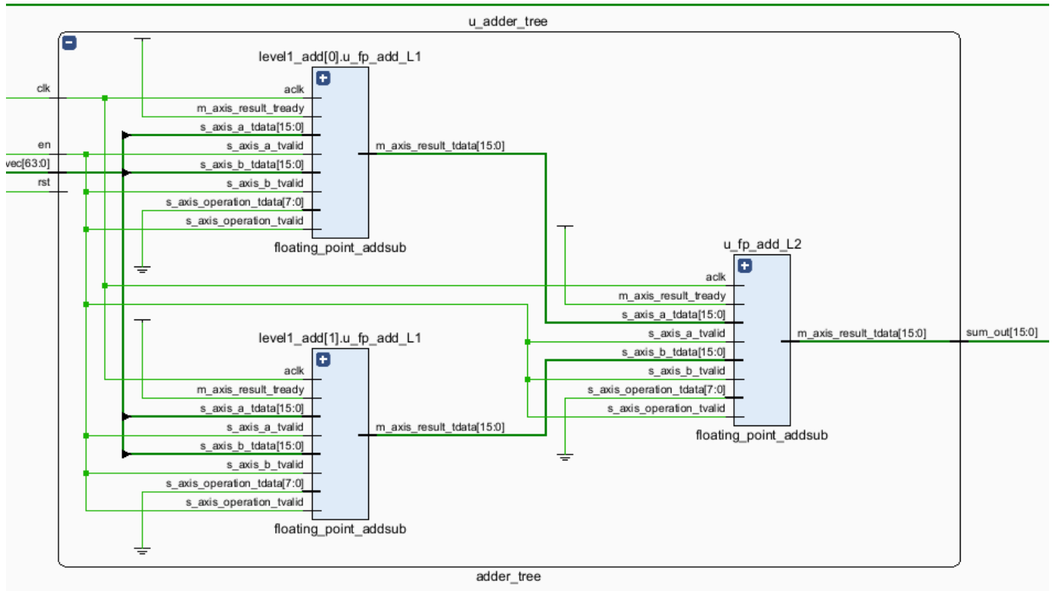
Schematic of SFU





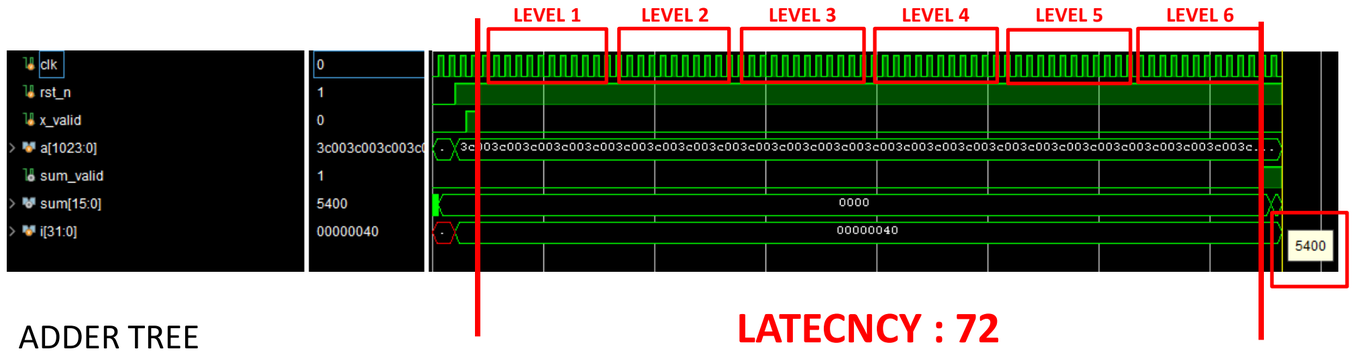
Schematic of ADDER TREE





Check the IP Catalog Latency

Module	IP	latency
ADD(SUB)	Using floating_point_(add)	12
ADDER TREE	**Using floating_point_(add)**	**12 * 6 (6 Level : 64 → 32 .. →1)**
MULT	Using floating_point_(mult)	7
MULT 1/N	floating_point_(div)	16
RECIP_SQRT	floating_point_(rsqrt)	5



ADDER TREE

LATECNCY : 72

```
module adder_tree #(
    parameter N = 4,
    parameter WIDTH = 16
)(
    input  wire clk,
    input  wire rst,
    input  wire en,
    input  wire [N*WIDTH-1:0] in_vec,
    output wire [WIDTH-1:0] sum_out
);

    wire [WIDTH-1:0] in_data [0:N-1];
    genvar i;
    generate
        for (i = 0; i < N; i = i + 1) begin : input_split
            assign in_data[i] = in_vec[i*WIDTH +: WIDTH];
        end
    endgenerate

    wire [WIDTH-1:0] level_1 [0:(N/2)-1];
    wire [WIDTH-1:0] level_2;
```

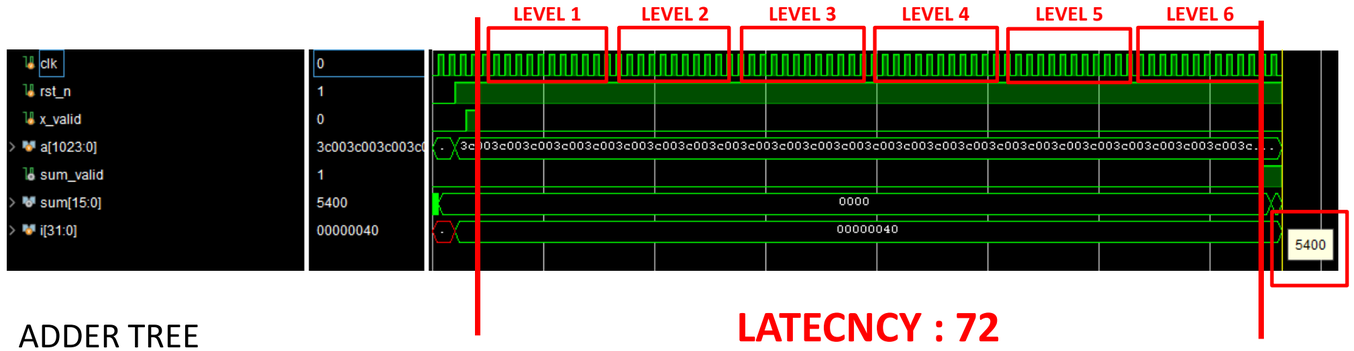
```
    // 1st LEVEL(N=4 : 4 → 2)
    generate
        for (i = 0; i < N/2; i = i + 1) begin : level1_add
            floating_point_addsub u_fp_add_L1 (
                .aclk(clk),
                .s_axis_a_tvalid(en),
                .s_axis_a_tdata(in_data[2*i]),
                .s_axis_b_tvalid(en),
                .s_axis_b_tdata(in_data[2*i+1]),
                .s_axis_operation_tvalid(en),
                .s_axis_operation_tdata(2'b00), // ADD
                .m_axis_result_tvalid(),
                .m_axis_result_tready(1'b1),
                .m_axis_result_tdata(level_1[i])
            );
        end
    endgenerate
```

```
    // 2nd LEVEL(2 → 1)
    floating_point_addsub u_fp_add_L2 (
        .aclk(clk),
        .s_axis_a_tvalid(en),
        .s_axis_a_tdata(level_1[0]),
        .s_axis_b_tvalid(en),
        .s_axis_b_tdata(level_1[1]),
        .s_axis_operation_tvalid(en),
        .s_axis_operation_tdata(2'b00),
        .m_axis_result_tvalid(),
        .m_axis_result_tready(1'b1),
        .m_axis_result_tdata(level_2)
    );

    assign sum_out = level_2;
```

Apply **ADDER TREE** instead Accumulator
But Using the **Pairwise** to calculate variance and
also using **Shifter** reason why **denominator is power 2**

Check the IP Catalog Latency

Module	IP	latency
ADD(SUB)	Using floating_point_(add)	12
ADDER TREE	**Using floating_point_(add)**	**12 * 6 (6 Level : 64 → 32 .. →1)**
MULT	Using floating_point_(mult)	7
MULT 1/N	floating_point_(div)	16
RECIP_SQRT	floating_point_(rsqrt)	5



ADDER TREE

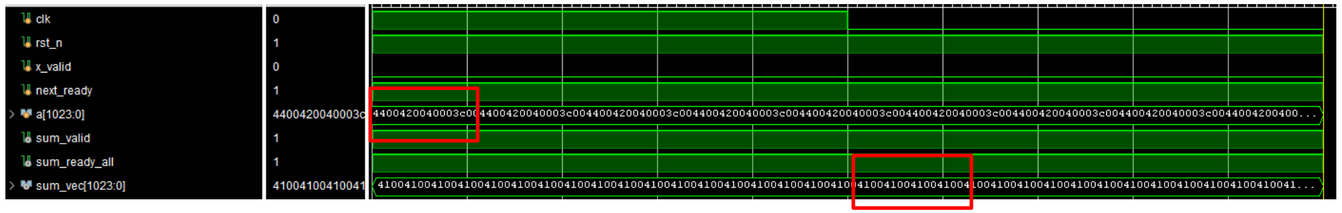
LATECNCY : 72

ADDER TREE	
Input A[63:0]	3c00(1)
Result	5400(64)

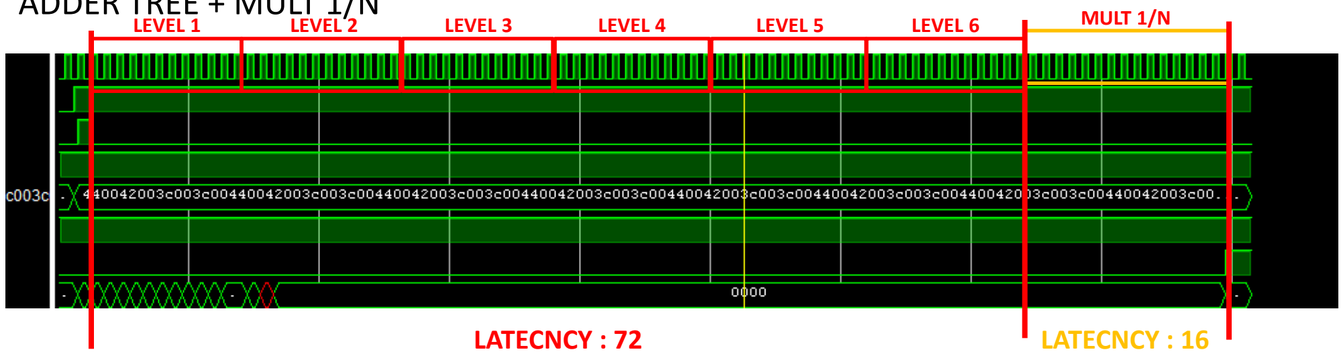
Apply **ADDER TREE** instead Accumulator
But Using the **Pairwise** to calculate variance and
also using **Shifter** reason why **denominator is power 2**

Check the IP Catalog Latency

Module	IP	latency
ADD(SUB)	Using floating_point_(add)	12
ADDER TREE	**Using floating_point_(add)**	**12 * 6 (6 Level : 64 → 32 .. →1)**
MULT	Using floating_point_(mult)	7
MULT 1/N	**floating_point_(div)**	**16**
RECIP_SQRT	floating_point_(rsqrt)	5



ADDER TREE + MULT 1/N



LATECNCY : 88

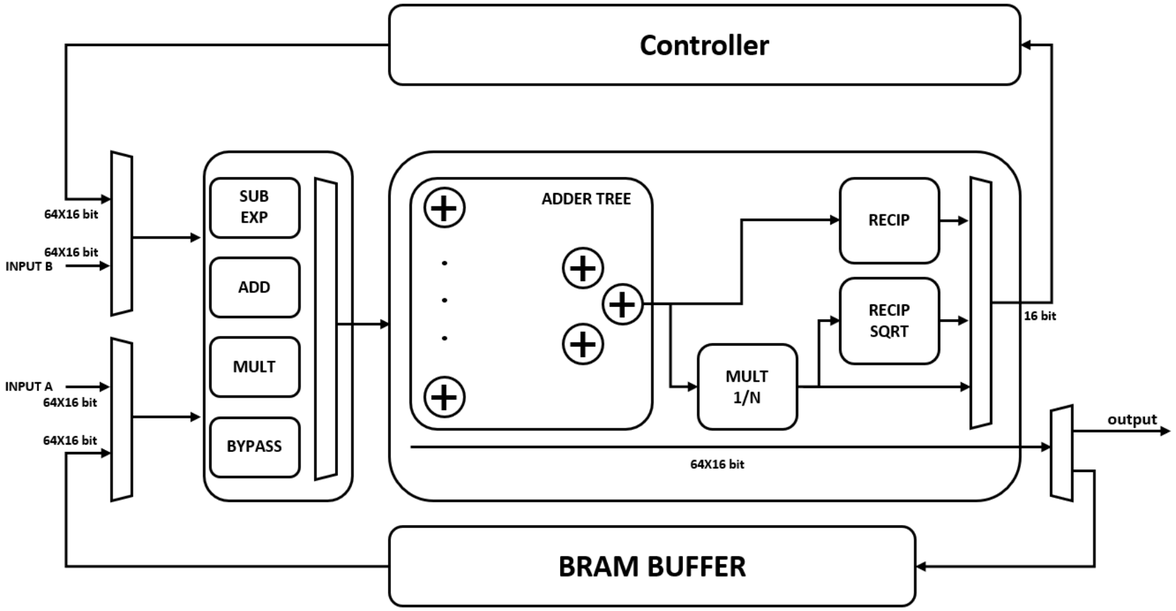
adder_div module code

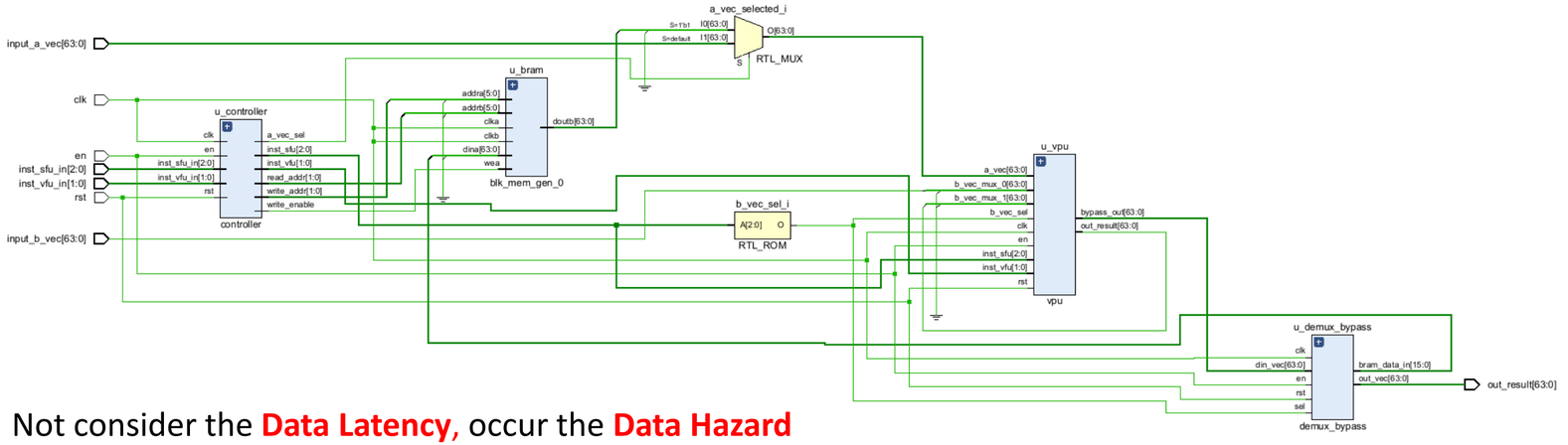
```
wire div_ready;
assign div_ready = downstream_ready;  // downstream이 ready이면 divider도 ready
assign input_ready = sum_ready_all & div_ready;

adder u_adder (
    .clk(clk),
    .rst_n(rst_n),
    .x(a_vec),
    .x_valid(in_valid),
    .sum_ready_all(sum_ready_all),
    .next_ready(div_ready),          // send to sum when ready to divider
    .sum_valid(sum_valid),
    .sum(adder_sum)
);

// 2. Divide sum by constant 64.0
wire [15:0] const_64_fp16 = 16'h5400;  // 64.0 in FP16

floating_point_div u_div (
    .aclk(clk),
    .s_axis_a_tvalid(sum_valid),
    .s_axis_a_tdata(adder_sum),
    .s_axis_b_tvalid(1'b1),
    .s_axis_b_tdata(const_64_fp16),
    .m_axis_result_tvalid(out_valid),
    .m_axis_result_tdata(mean),
    .m_axis_result_tready(div_ready)  // output result when the divider is downstream_ready
);
```

ADDER TREE + MULT 1/N				
Input A	4400(4) * 8	4200(3) * 8	4000(2) * 8	3c00(1) *
Result	4100(2.5) := (4+3+2+1) * 8 / 64			

Apply **ADDER TREE** instead Accumulator
But Using the **Pairwise** to calculate variance and
also using **Shifter** reason why **denominator is power 2**

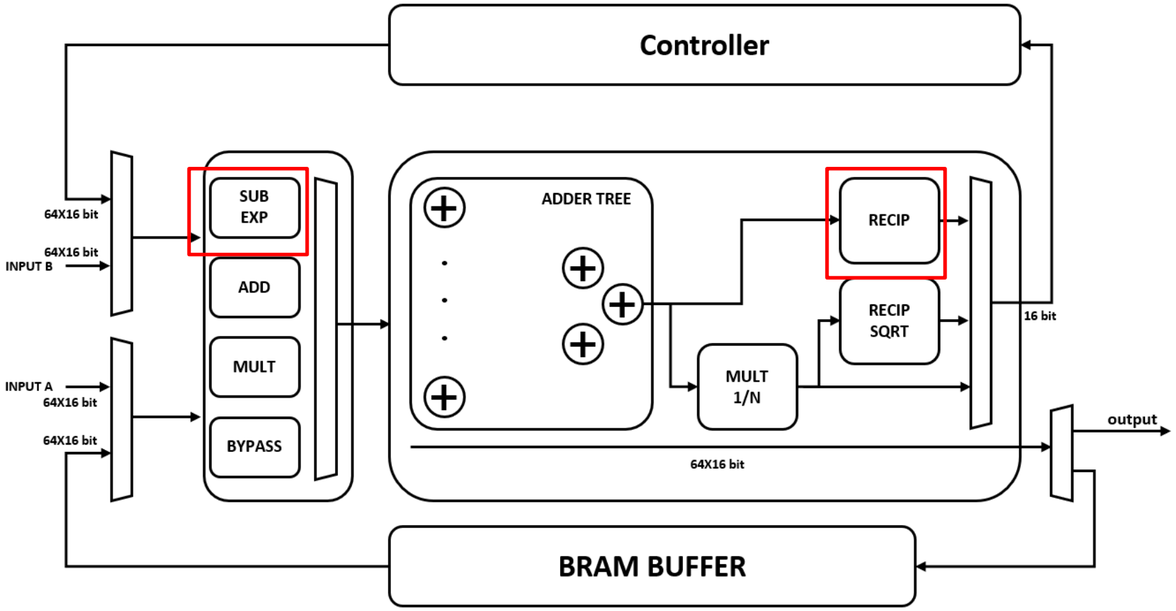
Schematic of TOP_module





Not consider the **Data Latency**, occur the **Data Hazard**

VPU for LayerNorm Acceleration



Not Using the **SUB &EXP, RECIP**

reason why using Softmax

Separate 7 **Stage**

1. Store the **INPUT data** $X_i[63:0]$ where BRAM BUFFER

2. Calculate **mean** $\mu_x[63:0]$

3. Substrate $X_i[63:0] - \mu_x[63:0]$

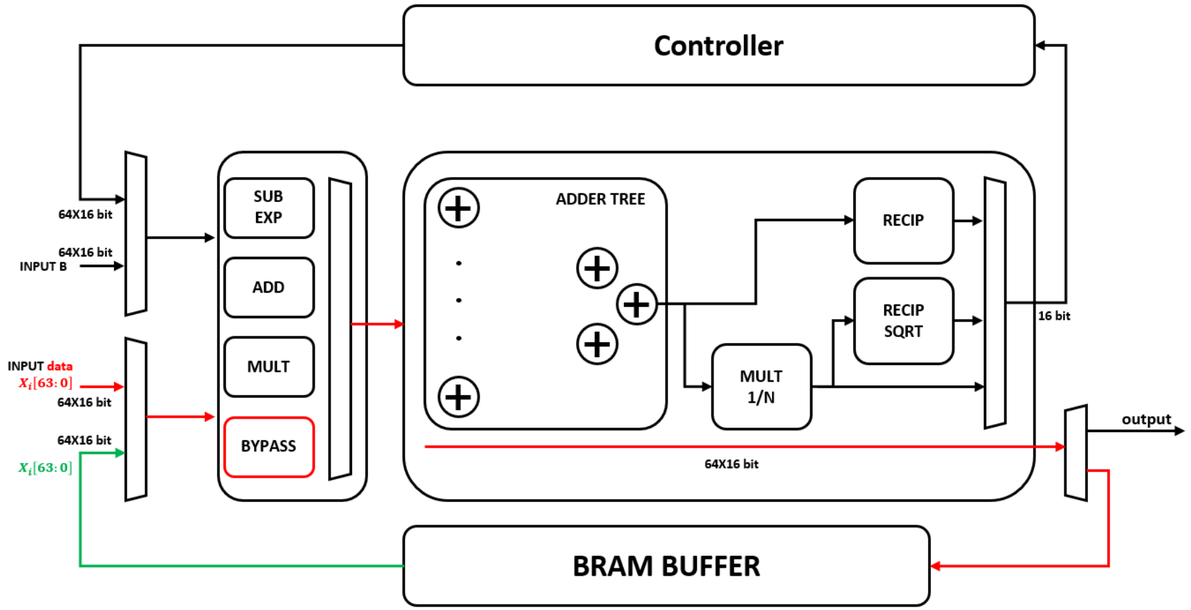
4. Calculate $\frac{1}{\sigma_x[63:0]}$

5. Mult $\frac{1}{\sigma_x[63:0]}$ and $X_i[63:0] - \mu_x[63:0]$ to Normalize

6. Mult **Weight**

7. Add **bias**

VPU for LayerNorm Acceleration



Module	IP	latency
Block memory Generator	BRAM BUFFER	-

Operation

VFU – bypass

SFU – bypass

store the **BRAM BUFFER**

Stage1 Result : $\boldsymbol{X_i[63:0]}$

VPU for LayerNorm Acceleration

Stage 1

```
module LN_stage12(
    input  wire clk,
    input  wire rst_n,

    input  wire x_valid,            // Set x_valid when Input data valid
    input  wire [64*16-1:0] a,      // Input data 64EA * 16bit(IEEE)

    input  wire downstream_ready,   // Consider to IP latency
    output wire input_ready,        // Consider to IP latency

    output reg  [15:0] mean_out,    // Result of output value -> mean
    output reg  mean_valid          // Check the output data is valid
);

    // Stage1 : Store the input value where the register when both x_valid & input_ready
    reg [64*16-1:0] reg_a;          // input value data is 64EA * 16bit
    reg stage1_valid;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            reg_a        <= {64*16{1'b0}};
            stage1_valid <= 1'b0;
        end else if (x_valid && input_ready) begin  // when x_valid and input_ready is 1
            reg_a        <= a;                      // Store the value where the register
            stage1_valid <= 1'b1;                   // Set stage1_valid => 1
        end else begin
            stage1_valid <= 1'b0;
        end
    end
```

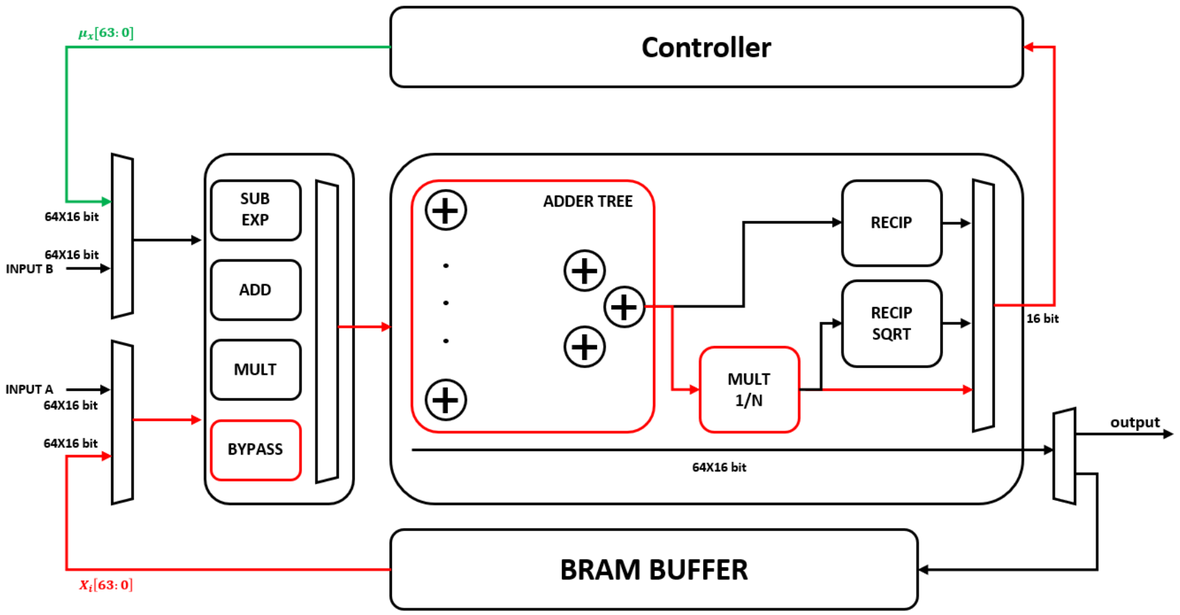
Input data_tb

```
// prepare input vector pattern: 1.0,2.0,4.0,8.0 repeating
for (i = 0; i < 64; i = i + 1) begin
  case (i % 4)
    0: a[i*16 +:16] = 16'h3C00; // 1.0
    1: a[i*16 +:16] = 16'h4000; // 2.0
    2: a[i*16 +:16] = 16'h4400; // 4.0
    3: a[i*16 +:16] = 16'h4800; // 8.0
  endcase
end
```

	Input 1	Input 2	Input 3	Input 4
Input A	4800(8.0)	4400(4.0)	4000(2.0)	3C00(1.0)
Result	**4800(8.0)**	**4400(4.0)**	**4000(2.0)**	**3C00(1.0)**

Stage1 Result : $\boldsymbol{X_i[63:0]}$

VPU for LayerNorm Acceleration



Module	IP	latency
ADDER TREE	Using floating_point_(add)	12 * 6 (6 Level : 64 → 32 .. →1)
MULT 1/N	floating_point_(div)	16

Operation

VFU – bypass

SFU – ADDER TREE = $\sum_{i=0}^{63} X_i$

→ Div(Mult 1/N) = μ_x

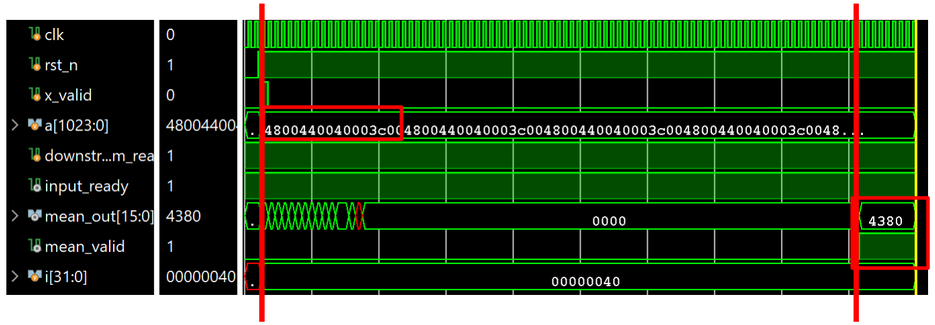
Broadcast **Scalar to Vector**

Stage2 Result : $\mu_x[63:0]$

VPU for LayerNorm Acceleration

Stage 2

```
    // Stage2 : Calculate the total sum of the input value and divide the N(64)
    wire div_ready;
    wire div_valid;
    wire [15:0] div_mean;
    adder_div u_div (
        .clk              (clk),
        .rst_n            (rst_n),
        .in_valid         (stage1_valid),       // stage1_valid -> set 1 where stage 1
        .a_vec            (reg_a),              // input value -> reg_a
        .downstream_ready (downstream_ready),   // Check the next stae is Ready
        .out_valid        (div_valid),          // output the 1 when done the calculate adder_div
        .mean             (div_mean),           // result of calculate Adder_Tree and divide
        .input_ready      (div_ready)           // Ready to input the data reg_a
    );

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            mean_out   <= 16'd0;
            mean_valid <= 1'b0;
        end else begin
            mean_out   <= div_mean;             // result : mean
            mean_valid <= div_valid;            // valid the value when mean_valid is 1
        end
    end

    // back-pressure handshake
    assign input_ready = div_ready;

endmodule
```



data

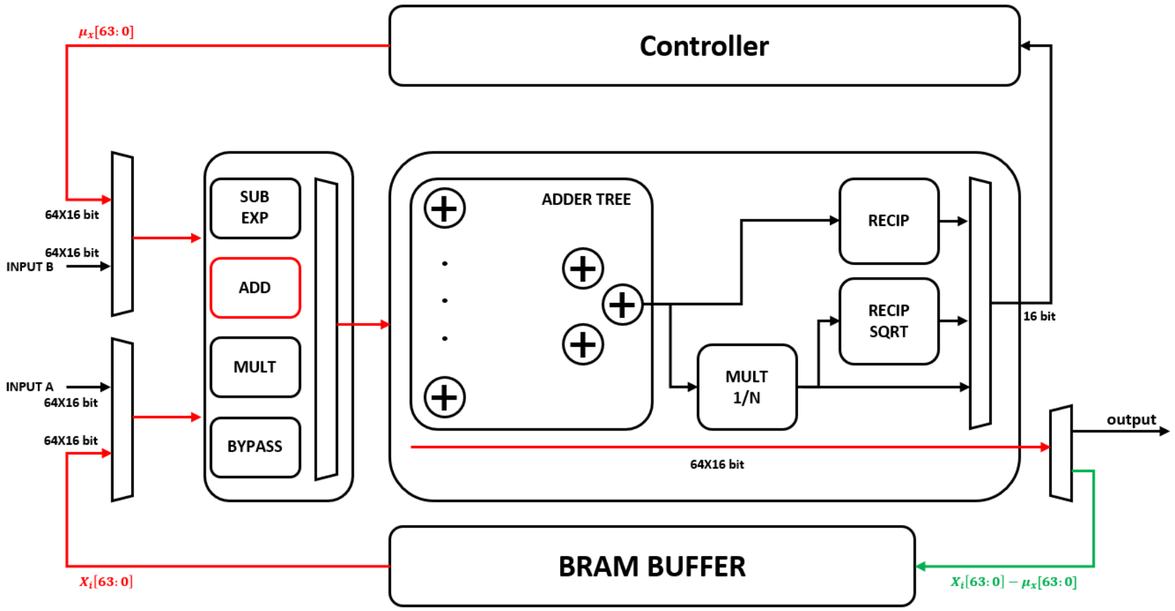
Input A : reg_a(BRAM)

	Input 1	Input 2	Input 3	Input 4
Input A	4800(8.0)	4400(4.0)	4000(2.0)	3C00(1.0)
Result	**4380(3.75)**			

$$(8 + 4 + 2 + 1) \times {}^{16}/_{64} = 3.75$$

Stage2 Result : $\boldsymbol{\mu_x = 4380(3.75)}$

VPU for LayerNorm Acceleration



Module	IP	latency
ADD(SUB)	Using floating_point_(add)	12

Operation

VFU – ADD(SUB) = $X_i[63:0] - \mu_x[63:0]$

SFU – bypass

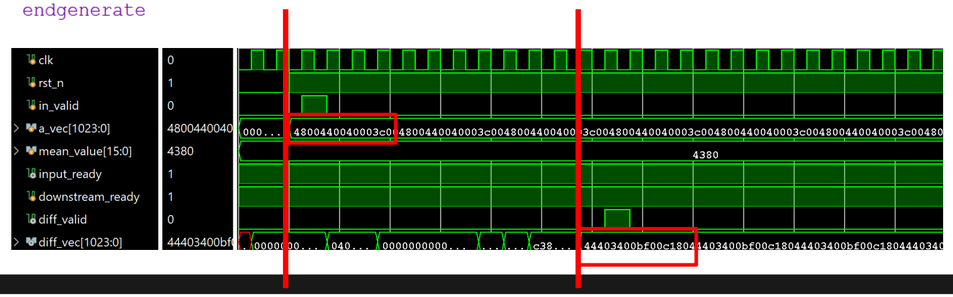
store the **BRAM BUFFER**

Stage3 Result : $X_i[63:0] - \mu_x[63:0]$

VPU for LayerNorm Acceleration

Stage 3

```
// per-lane ready/valid vectors
wire [63:0] sub_a_ready, sub_b_ready, diff_valid_vec;
wire [64*16-1:0] diff_bus;

// Stage3(sub)'s ready & Stage 4's Ready state
assign input_ready = (&sub_a_ready) && (&sub_b_ready) && downstream_ready;

// diff_valid: all channel's result valid
assign diff_valid    = &diff_valid_vec;     // sent to Stage 4 valid -> 1
assign diff_vec      = diff_bus;            // result of Input A and B's diff
assign dbg_sub_valid = diff_valid_vec;

genvar gi;
generate
  for (gi = 0; gi < 64; gi = gi + 1) begin : GEN_SUB
    addsub u_sub (
      .clk           (clk),
      .rst_n         (rst_n),
      .valid_in      (in_valid),          // Stage2(mean)'s valid
      .add_en        (1'b0),              // subtraction mode
      .a             (a_vec[gi*16 +:16]), // input A : input vector value
      .b             (mean_value),        // input B : μ_x
      .valid_out     (diff_valid_vec[gi]),
      .ready_in      (downstream_ready),
      .result        (diff_bus[gi*16 +:16]), // X_i [63:0]-μ_x [63:0]
      .ready_a       (sub_a_ready[gi]),
      .ready_b       (sub_b_ready[gi])
    );
  end
endgenerate
```



data

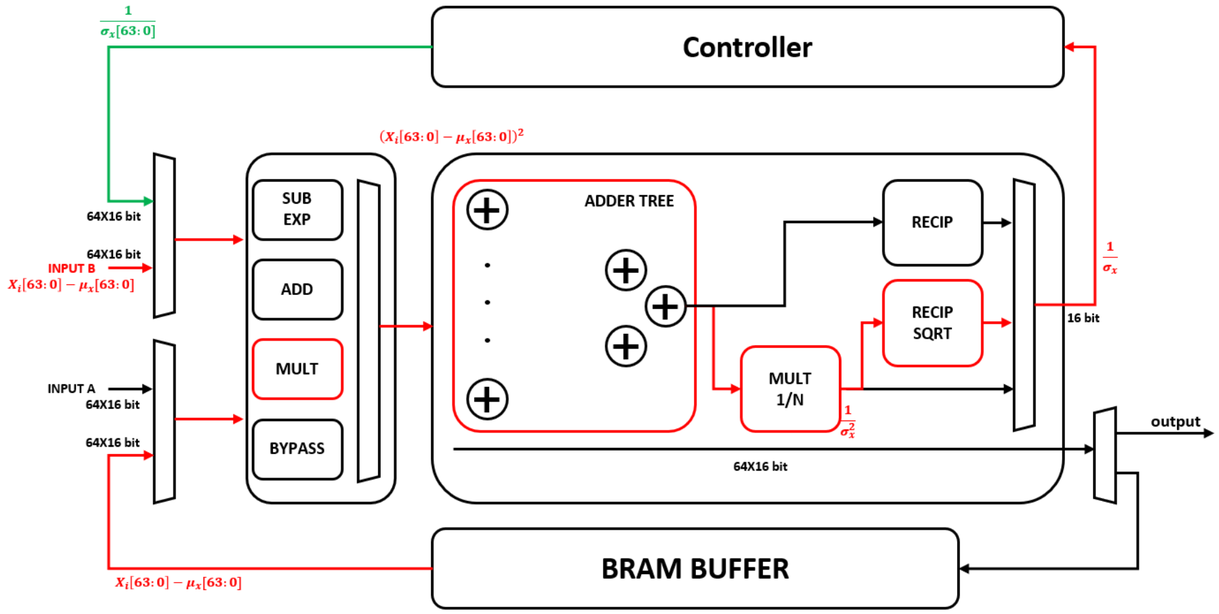
Input A : reg_a(BRAM)

Input B : $\mu_x[63:0] := 4380(3.75)$

	Input 1	Input 2	Input 3	Input 4
Input A	4800(8.0)	4400(4.0)	4000(2.0)	3C00(1.0)
Input B	4380(3.75)	4380(3.75)	4380(3.75)	4380(3.75)
Result	**4400(4.25)**	**3400(0.25)**	**BF00(-1.75)**	**C180(-2.75)**

Stage3 Result : $X_i[63:0] - \mu_x[63:0]$

VPU for LayerNorm Acceleration



Module	IP	latency
MULT	Using floating_point_(mult)	7
ADDER TREE	Using floating_point_(add)	72(:= 12*6)
MULT 1/N	floating_point_(div)	16
RECIP_SQRT	floating_point_(rsqrt)	5

Operation

VFU – MULT = $(X_i[63:0]-\mu_x[63:0])^2$

SFU – ADDER TREE = $\sum_{i=0}^{63} X_i - \mu_x$

→ **Div(Mult 1/N) =** $\frac{1}{\sigma_x^2}$

→ **RECIP_SQRT =** $\frac{1}{\sigma_x}$

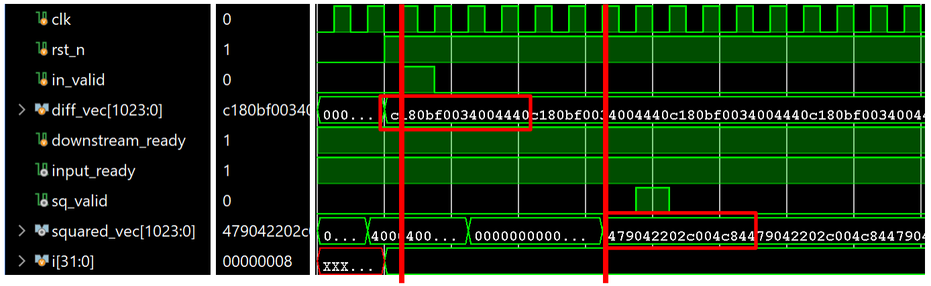
Broadcast **Scalar to Vector**

Stage4 Result : $\frac{1}{\sigma_x[63:0]}$

VPU for LayerNorm Acceleration

Stage 4-1

```
// Stage4 & Stage 5 Ready state
assign input_ready  = (&a_ready) & (&b_ready) & downstream_ready;
// valid when all lanes have produced result
assign sq_valid     = &sq_valid_vec;
assign squared_vec  = sq_bus;
assign dbg_sq_valid = sq_valid_vec;

genvar i;
generate
  for (i = 0; i < 64; i = i + 1) begin : GEN_SQ
    floating_point_mult u_sq (
      .aclk                 (clk),

      .s_axis_a_tvalid      (in_valid),
      .s_axis_a_tdata       (diff_vec[i*16 +:16]),    // input diff_vec(vec - mean)
      .s_axis_a_tready      (a_ready[i]),
      .s_axis_b_tvalid      (in_valid),
      .s_axis_b_tdata       (diff_vec[i*16 +:16]),    // input diff_vec(vec - mean)
      .s_axis_b_tready      (b_ready[i]),

      .m_axis_result_tvalid (sq_valid_vec[i]),
      .m_axis_result_tdata  (sq_bus[i*16 +:16]),      // Result : (X_i [63:0]-μ_x [63:0])^2
      .m_axis_result_tready (downstream_ready)
    );
  end
endgenerate
```



data

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

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

	Input 1	Input 2	Input 3	Input 4
Input A	4400(4.25)	3400(0.25)	BF00(-1.75)	C180(-2.75)
Input B	4400(4.25)	3400(0.25)	BF00(-1.75)	C180(-2.75)
Result	**4C84(18.0625)**	**2C00(0.0625)**	**4220(3.0625)**	**4790(7.5625)**

Stage4-1 Result : $(X_i[63:0] - \mu_x[63:0])^2$

VPU for LayerNorm Acceleration

Stage 4-2

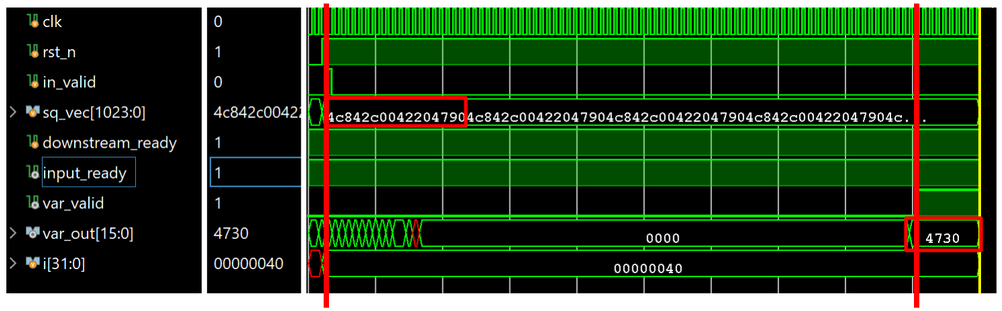
```
adder_div u_var (
   .clk              (clk),
   .rst_n            (rst_n),
   .in_valid         (in_valid),
   .a_vec            (sq_vec),          // (X_i [63:0]-μ_x [63:0])^2
   .downstream_ready (downstream_ready),
   .out_valid        (var_valid),
   .mean             (var_out)          // 1/(σ_x^2 )
);
```

data

Input A : $(X_i[63:0] - \mu_x[63:0])^2$

	Input 1	Input 2	Input 3	Input 4
Input A	4C84(18.0625)	2C00(0.0625)	4220(3.0625)	4790(7.5625)
Result	**4730(7.1875)**			

$$(18.0625 + 0.0625 + 3.0625 + 7.5625) \times {}^{16}/_{64} = 7.1875$$

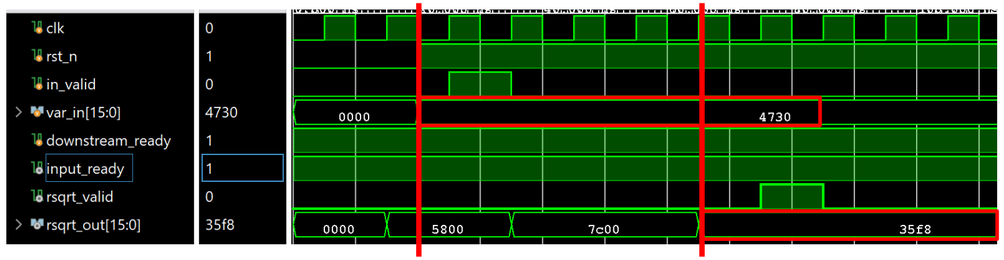


Stage4-2 Result : $\frac{1}{\sigma_x^2}$

VPU for LayerNorm Acceleration

Stage 4-3

```
wire a_ready;
wire result_ready = downstream_ready; // normalization always ready

floating_point_rsqrt u_rsqrt (
  .aclk                 (clk),
  .s_axis_a_tvalid      (in_valid),
  .s_axis_a_tdata       (var_in),          // 1/(σ_x^2 )
  .s_axis_a_tready      (a_ready),
  .m_axis_result_tvalid (rsqrt_valid),
  .m_axis_result_tdata  (rsqrt_out),       // 1/σ_x
  .m_axis_result_tready (result_ready)
);
```



data

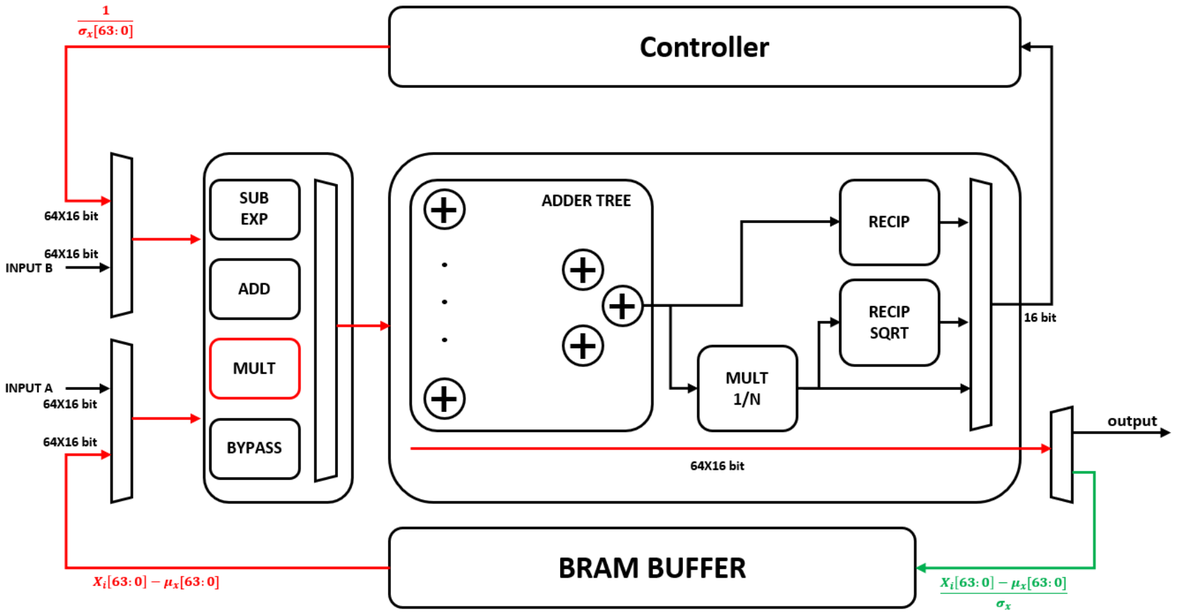
Input A : $\frac{1}{\sigma_x^2} := 4730(7.1875)$

	Input 1	Input 2	Input 3	Input 4
Input A	4730(7.1875)			
Result	35F8(0.3730)			

$$^{1}/_{\sqrt{7.1875}} = 0.3730$$

Stage4-3 Result : $\frac{1}{\sigma_x}$

VPU for LayerNorm Acceleration



Module	IP	latency
MULT	Using floating_point_(mult)	7

Operation

VFU – MULT = $\frac{X_i[63:0]-\mu_x[63:0]}{\sigma_x}$

SFU – bypass

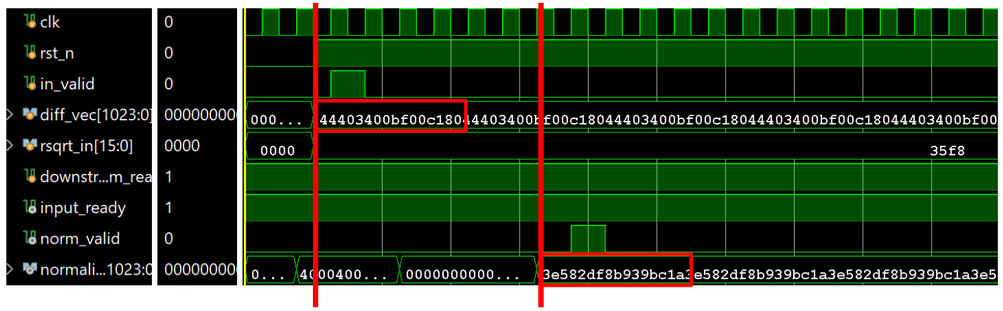
store the **BRAM BUFFER**

Stage5 Result : $\frac{X_i[63:0]-\mu_x[63:0]}{\sigma_x}$

VPU for LayerNorm Acceleration

Stage 5

```
assign input_ready    = (&a_ready) & (&b_ready) & downstream_ready;
assign norm_valid     = &norm_valid_vec;
assign normalized_vec = norm_bus;

genvar i;
generate
  for (i = 0; i < 64; i = i + 1) begin : GEN_NORM
    floating_point_mult u_norm (
      .aclk                 (clk),
      .s_axis_a_tvalid      (in_valid),
      .s_axis_a_tdata       (diff_vec[i*16 +:16]),    // input A : X_i [63:0]-μ_x [63:0]

      .s_axis_a_tready      (a_ready[i]),
      .s_axis_b_tvalid      (in_valid),
      .s_axis_b_tdata       (rsqrt_in),               // input B : 1/σ_x
      .s_axis_b_tready      (b_ready[i]),
      .m_axis_result_tvalid (norm_valid_vec[i]),
      .m_axis_result_tdata  (norm_bus[i*16 +:16]),    // result : (X_i [63:0]-μ_x [63:0])/σ_x
      .m_axis_result_tready (downstream_ready)
    );
  end
endgenerate
```



data

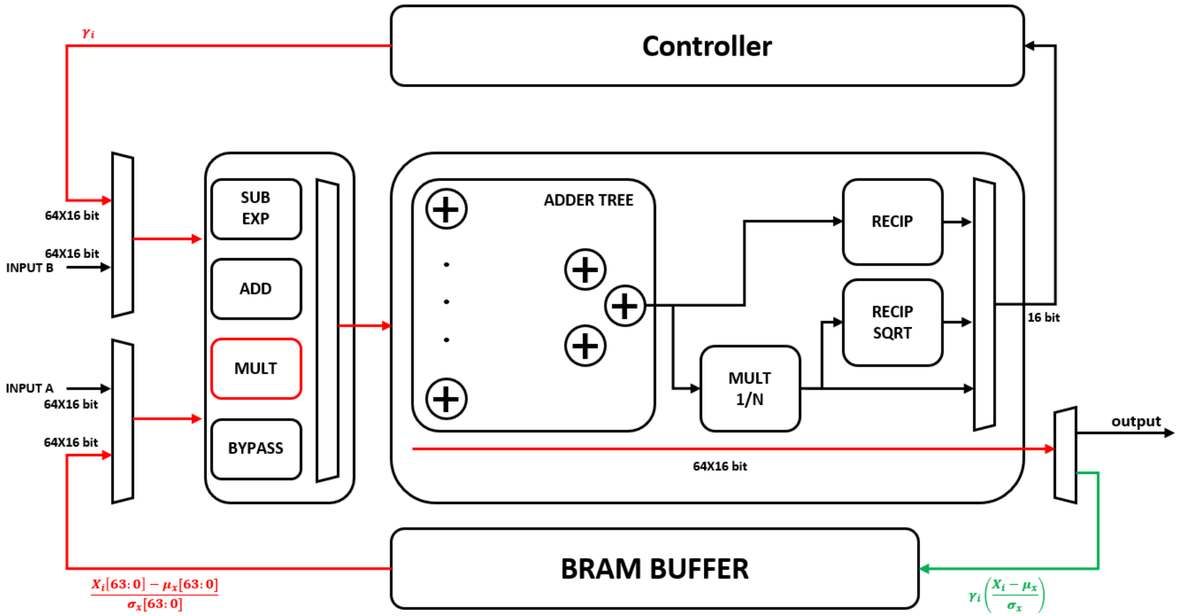
Input A : $X_i[63:0] - \mu_x[63:0]$**(BRAM)**

Input B : $\frac{1}{\sigma_x} \coloneqq \mathbf{35F8(0.3730)}$

	Input 1	Input 2	Input 3	Input 4
Input A	4400(4.25)	3400(0.25)	BF00(-1.75)	C180(-2.75)
Input B	35F8(0.3730)			
Result	**3E58(1.59)**	**2DF8(0.09)**	**B939(-0.65)**	**BC1A(-1.03)**

Stage5 Result : $\frac{X_i[63:0]-\mu_x[63:0]}{\sigma_x}$

VPU for LayerNorm Acceleration



Module	IP	latency
MULT	Using floating_point_(mult)	7

Operation

VFU – MULT = $\boldsymbol{\gamma_i\left(\frac{X_i-\mu_x}{\sigma_x}\right)}$

SFU – bypass

store the **BRAM BUFFER**

Stage6 Result : $\boldsymbol{\gamma_i\left(\frac{X_i-\mu_x}{\sigma_x}\right)}$

VPU for LayerNorm Acceleration

Stage 6

```
genvar i;
generate
  for(i = 0; i < 64; i = i + 1) begin : GEN_MUL
    floating_point_mult u_mul (
      .aclk                 (clk),
      .s_axis_a_tvalid      (in_valid),
      .s_axis_a_tdata       (normalized_vec[i*16 +:16]),  // Input A : Result of Stage 5 Norm
      .s_axis_a_tready      (),
      .s_axis_b_tvalid      (in_valid),
      .s_axis_b_tdata       (weight),                     // Input B : weight => 4000(2)
      .s_axis_b_tready      (),
      .m_axis_result_tvalid (mult_valid_vec[i]),
      .m_axis_result_tdata  (mult_bus[i*16 +:16]),        // Result : γ_i ((X_i-μ_x)/σ_x )
      .m_axis_result_tready (1'b1)
    );
  end
endgenerate
```

data

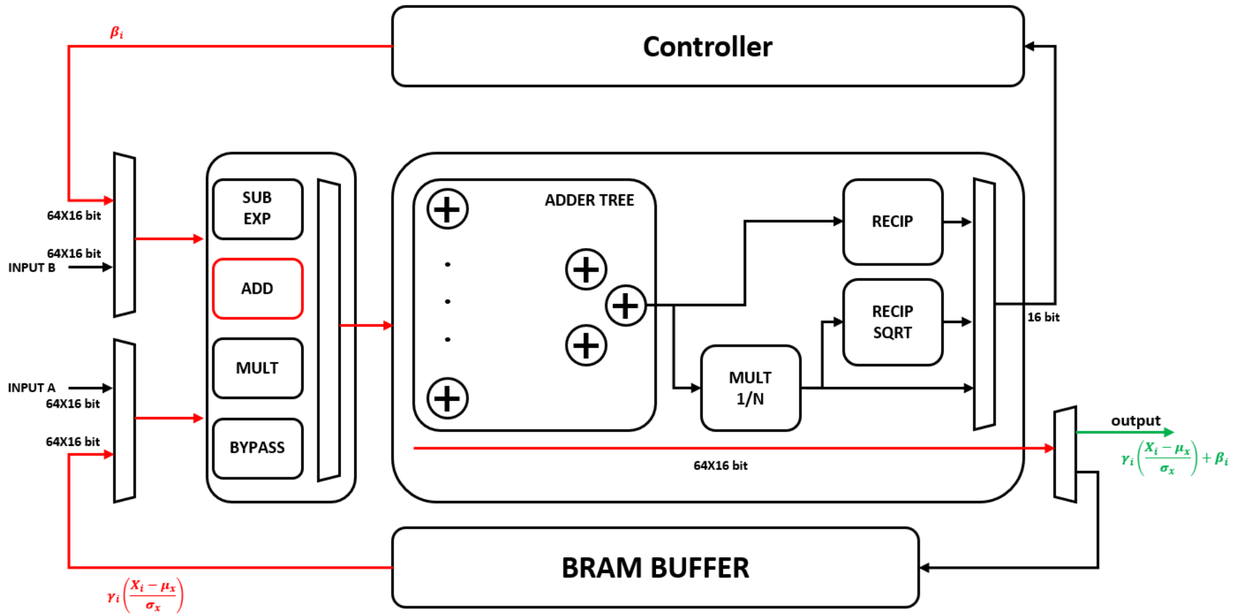
Input A : $\gamma_i \left(\frac{X_i - \mu_x}{\sigma_x}\right)$

Input B : bias $\gamma_i := 4000\ (2)$

	Input 1	Input 2	Input 3	Input 4
Input A	3E58(1.59)	2DF8(0.09)	B939(-0.65)	BC1A(-1.03)
Input B	4000(2)			

Stage6 Result : $\gamma_i \left(\frac{X_i - \mu_x}{\sigma_x}\right)$

VPU for LayerNorm Acceleration



Operation

VFU – ADD = $\boldsymbol{\gamma_i\left(\frac{X_i-\mu_x}{\sigma_x}\right)+\beta_i}$

SFU – bypass

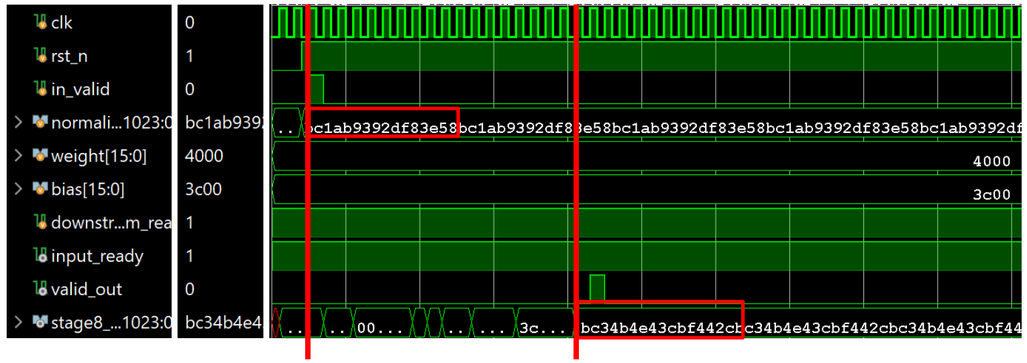
Module	IP	latency
ADD	Using floating_point_(add)	12

Stage7 Result : $\boldsymbol{\gamma_i\left(\frac{X_i-\mu_x}{\sigma_x}\right)+\beta_i}$

VPU for LayerNorm Acceleration

Stage 7

```
// 2) add bias
wire [63:0] add_valid_vec;

generate
  for(i = 0; i < 64; i = i + 1) begin : GEN_ADD
    addsub u_add (
      .clk        (clk),
      .rst_n      (rst_n),
      .valid_in   (mult_valid_vec[i]),
      .add_en     (1'b1),                    // Add mode
      .a          (mult_bus[i*16 +:16]),     // Input A : γ_i ((X_i-μ_x)/σ_x )
      .b          (bias),                    // Input B : bias => 3C00(1)
      .valid_out  (add_valid_vec[i]),
      .ready_in   (1'b1),
      .result     (stage8_out[i*16 +:16]),   // Result : γ_i ((X_i-μ_x)/σ_x )+β_i
      .ready_a    (),
      .ready_b    ()
    );
  end
endgenerate
```



data

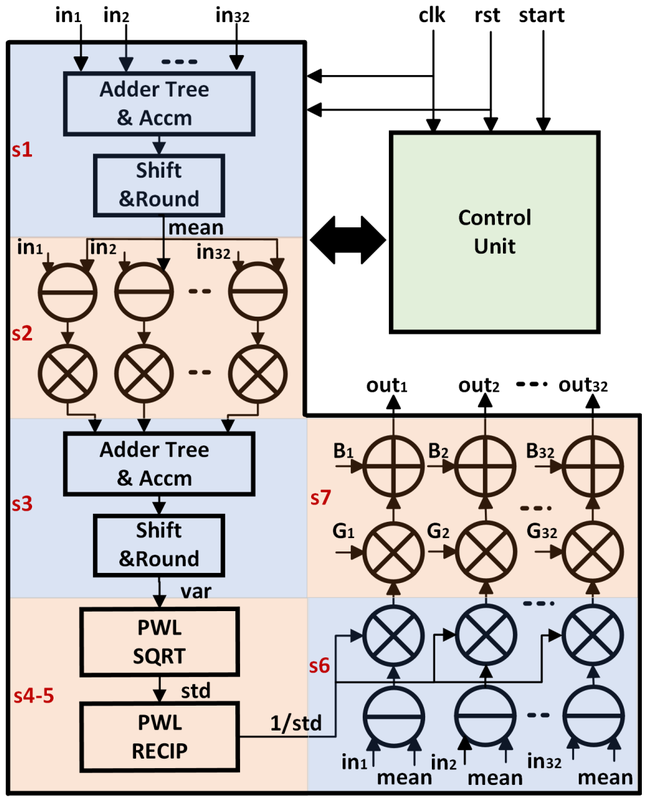
Input A : $\frac{X_i[63:0]-\mu_x[63:0]}{\sigma_x}$

Input B : bias $\beta_i := \mathbf{3C00}\ (1)$

	Input 1	Input 2	Input 3	Input 4
Input A	3E58(1.59)	2DF8(0.09)	B939(-0.65)	BC1A(-1.03)
Input γ_i	4000(2)			
Input B	3C00(1)			
Result	**442C(4.17)**	**3CBF(1.19)**	**B4E4(-0.31)**	**BC34(-1.05)**

Stage7 Result : $\gamma_i\left(\frac{X_i-\mu_x}{\sigma_x}\right)+\beta_i$

LN Approximation Model



Layer Normalization Approximation Architecture

1. Pairwise Module > compute mean, variance

(CLA adder, Mul, Adder Tree)

2. PWL SQRT/RECIP Module > approximate std, 1/std

(LUT, CLA adder, Mul)

3. Layer Normalization Module > compute $(x - \mu) * 1/std$

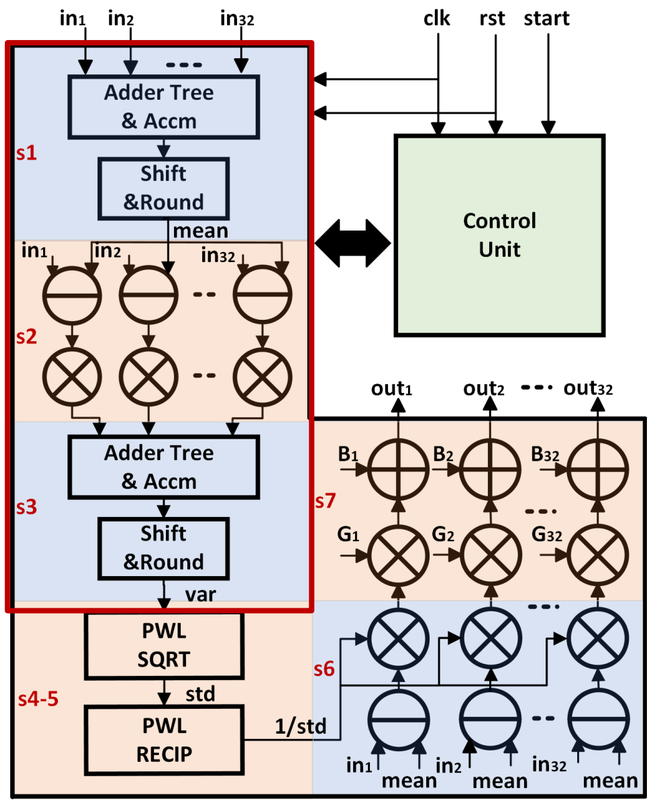
(CLA adder, Mul)

Input / Output Dimension (Current Test Setting)

- 256-dim Q8.8 format

(plan to scale down to 64 dimensions)

LN Approximation Model



Pairwise Module Architecture

1. Input Splitting & Group-wise Processing

- Total input: x_in[0] ~ x_in[255] (Q8.8 format)

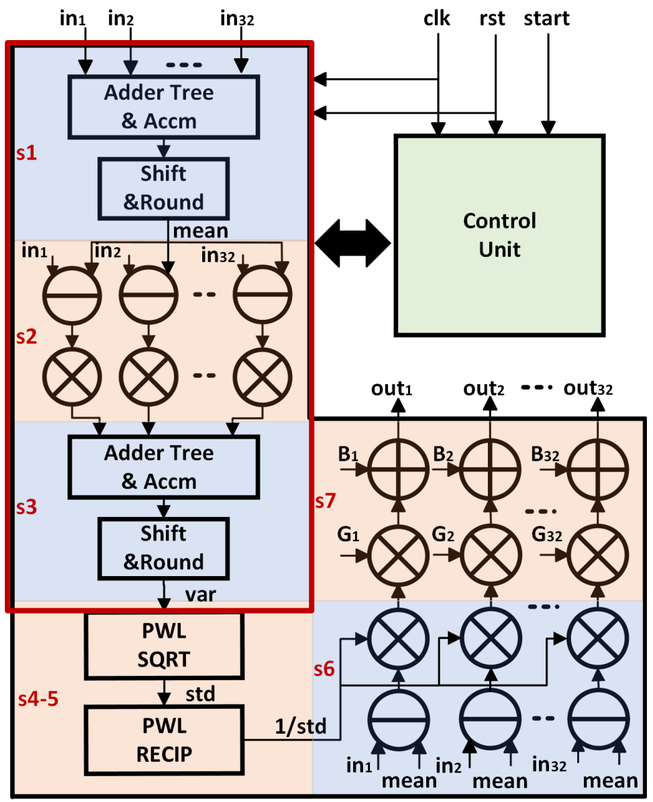
```
3       module pairwise_variance (
4           input  wire         clk,
5           input  wire         rst,
6           input  wire         valid_in,
7           input  wire [4095:0] x_in_flat, // 256 × 16bit
8           output reg          done,
9           output wire [15:0]  mean_q16,
0           output wire [15:0]  var_q16
1       );
```

- Group division: 16 groups (G = 16), each with 16 elements (N = 16)
- For each group:
  - Compute the group mean:

    group_mean[g] = sum(x_i for i in group g) >> 4
  - Compute group variance:

    group_M[g] = sum($(x_i - \mu)^2$ for i in group g)

LN Approximation Model



Pairwise Module Architecture

- Compute the group mean:

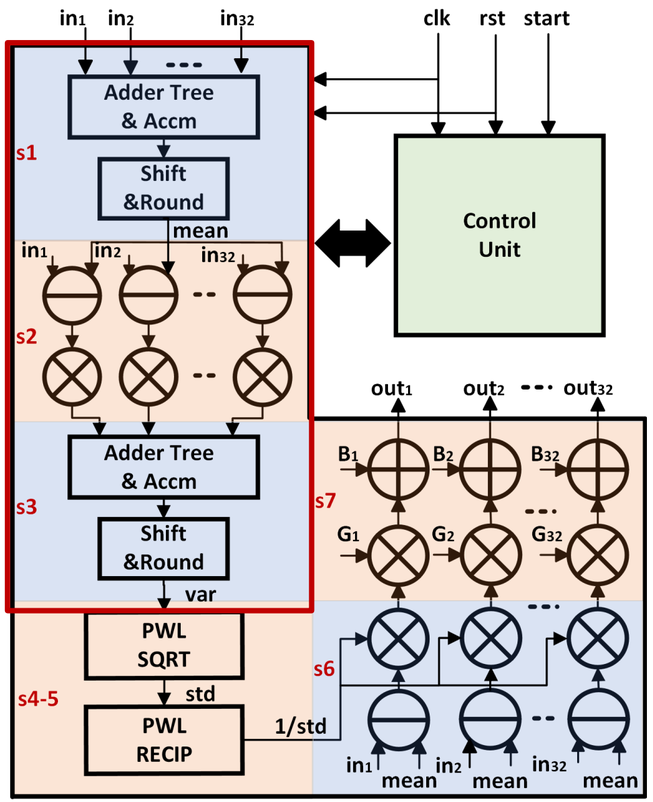
```
37          Tree_acc #(.N(16)) sum_tree (
38              .clk(clk), .rst(rst),
39              .x_in_flat({xi[15], xi[14], xi[13], xi[12], xi[11], xi[10], xi[9], xi[8],
40                          xi[7], xi[6], xi[5], xi[4], xi[3], xi[2], xi[1], xi[0]}),
41              .sum_out(sum)
42          );
43
44          wire [15:0] mu = sum >> 4;
45          assign group_mean[g] = mu;
```

- Compute group variance:

```
48          for (j = 0; j < 16; j = j + 1) begin : C_LOOP
49              wire [15:0] delta;
50              cla_adder sub (.a(xi[j]), .b(~mu + 1), .cin(1'b0), .sum(delta), .cout(cout));
51              signed_mul m1 (.clk(clk), .rst(rst), .a(delta), .b(delta), .result(d[j]));
52          end
53
54          Tree_acc #(.N(16)) M_tree (
55              .clk(clk), .rst(rst),
56              .x_in_flat({d[15], d[14], d[13], d[12], d[11], d[10], d[9], d[8],
57                          d[7], d[6], d[5], d[4], d[3], d[2], d[1], d[0]}),
58              .sum_out(group_M[g])
59          );
60      end
```

LN Approximation Model

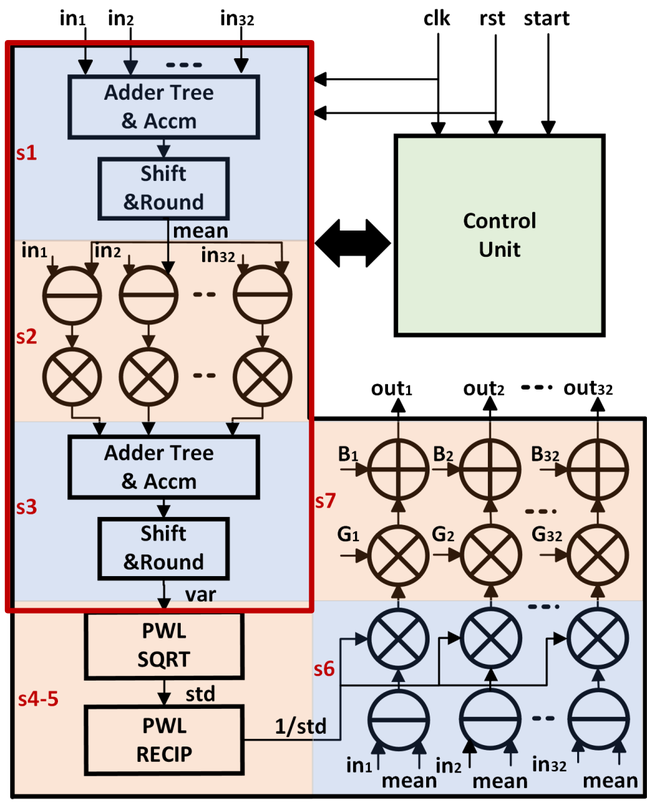
Pairwise Module Architecture



2. Global Mean Calculation

```
64      Tree_acc #(.N(16)) mean_tree (
65          .clk(clk), .rst(rst),
66          .x_in_flat({ group_mean[15], group_mean[14], group_mean[13], group_mean[12],
67                       group_mean[11], group_mean[10], group_mean[9],  group_mean[8],
68                       group_mean[7],  group_mean[6],  group_mean[5],  group_mean[4],
69                       group_mean[3],  group_mean[2],  group_mean[1],  group_mean[0] }),
70          .sum_out(sum_mu_total)
71      );
72
73      assign mean_q16 = sum_mu_total >> 4;
```

LN Approximation Model



Pairwise Module Architecture

3. Pairwise Variance Merging (LV1 → LV4)

- LV1: Merge pairs of groups (total 8 pairs)
- LV2: Merge M_lv1 (4 pairs)
- LV3: Merge M_lv2 (2 pairs)
- LV4: Final merge

→ computation

1. Mean difference

$$\delta = \mu_1 - \mu_2$$

2. Squared difference

$$\delta^2 = (\mu_1 - \mu_2) * (\mu_1 - \mu_2)$$

3. Scaled δ^2

$$\delta^2 \times \frac{n_1 + n_2}{n_1 * n_2}$$

4. Merged variance

$$M' = M_1 + M_2 + \delta^2$$

LN Approximation Model

Pairwise Module Architecture

3. Pairwise Variance Merging (LV1 → LV4)

- LV1: Merge pairs of groups (total 8 pairs)

```
79      for (k = 0; k < 8; k = k + 1) begin : LV1
80          wire [15:0] mu1 = group_mean[2*k];
81          wire [15:0] mu2 = group_mean[2*k+1];
82          wire [15:0] delta;
83          cla_adder sub (.a(mu1), .b(~mu2 + 1), .cin(1'b0), .sum(delta), .cout());
84          signed_mul m2 (.clk(clk), .rst(rst), .a(delta), .b(delta), .result(delta_sq1[k]));
85          signed_mul m3 (.clk(clk), .rst(rst), .a(delta_sq1[k]), .b(16'd256), .result(delta_term1[k]));
86          assign M_lv1[k] = group_M[2*k] + group_M[2*k+1] + (delta_term1[k] >> 5);
```

4. Final Variance Output

```
140     wire [15:0] M_total;
141     assign M_total = M_lv3[0] + M_lv3[1] + (delta_term_final >> 8);
142     assign var_q16 = M_total >> 8;
```

computation

1. Mean difference

$$\delta = \mu_1 - \mu_2$$

2. Squared difference

$$\delta^2 = (\mu_1 - \mu_2) * (\mu_1 - \mu_2)$$

3. Scaled δ^2

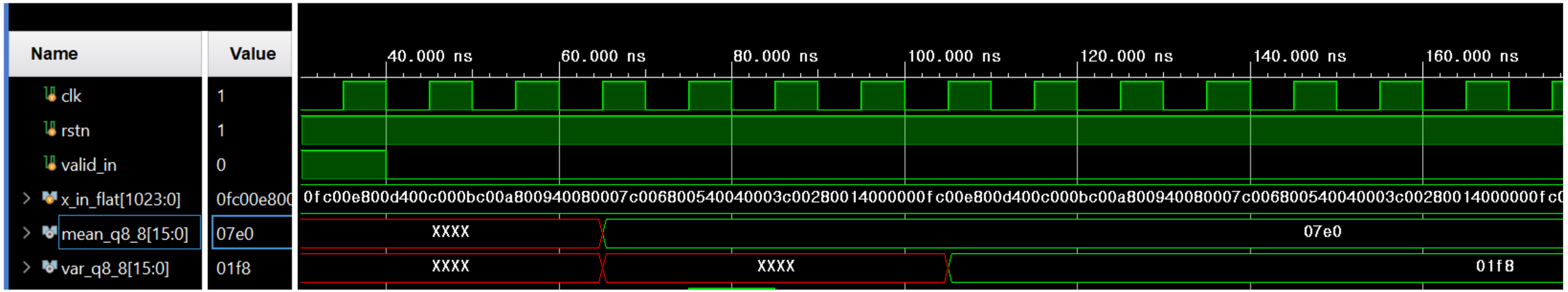
$$\delta^2 \times \frac{n_1 + n_2}{n_1 * n_2}$$

4. Merged variance

$$M' = M_1 + M_2 + \delta^2$$

LN Approximation Model

Pairwise Module Architecture

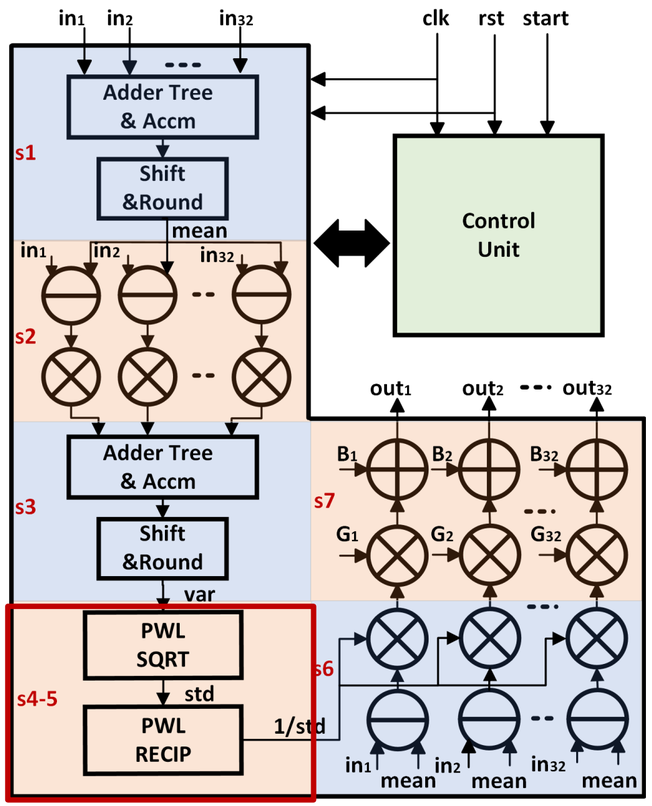


Mean(O)

Variance(X)

Variance is incorrect: expected 0x15F4, got 0x01F8.

LN Approximation Model



PWL Module Architecture

- **Input/Output**

  x_in : Input value (Q8.8 fixed-point)

  y_out: Output value (Q8.8 fixed-point)

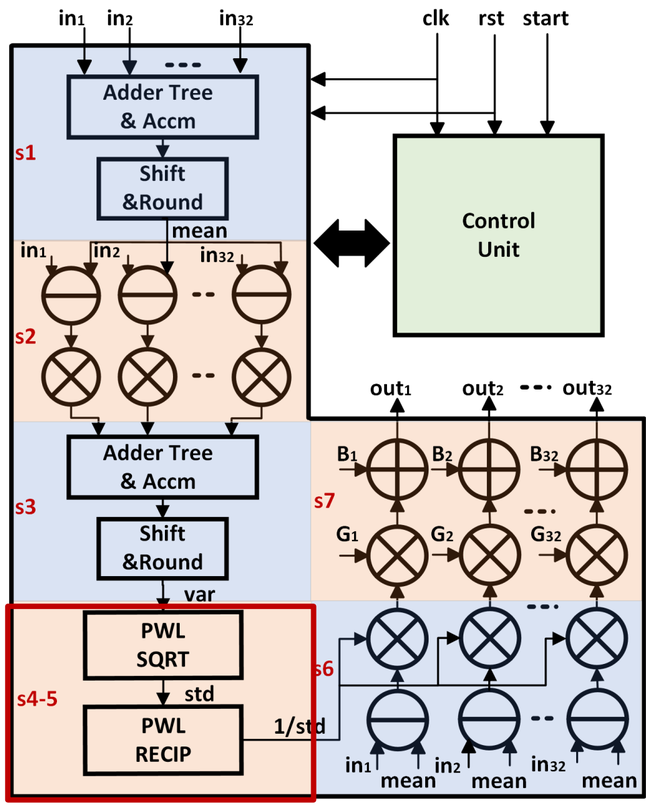
- **Breakpoints** {x0,x1,...,xn}: The domain segmentation
- **Slopes** {a0,a1,...,an−1}: Gradients of linear segments
- **Intercepts** {b0,b1,...,bn−1}: Y-intercepts of each line

(generated from NumPy .npz)

- breaks[i]≤x_in<breaks[i+1]

→ yout=slope[i]·x_in+intercept[i]

LN Approximation Model

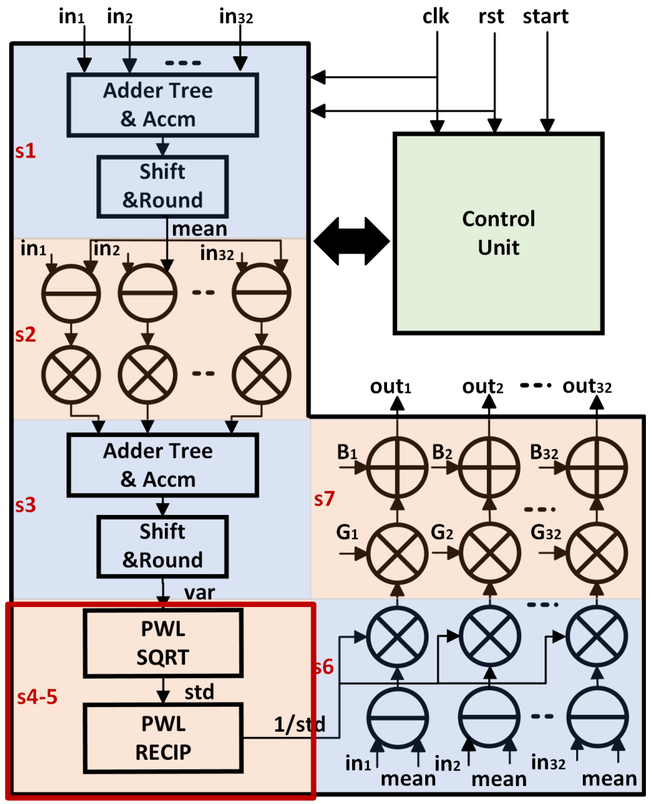


PWL Module Architecture

- breaks[i]≤xin<breaks[i+1]

```
53    always @(posedge clk) begin
54        for (j = 0; j < 8; j = j + 1) begin
55            if (x_in >= breaks[j] && x_in < breaks[j+1]) begin
56                seg_idx <= j[2:0];
57            end
58        end
59    end
```

LN Approximation Model



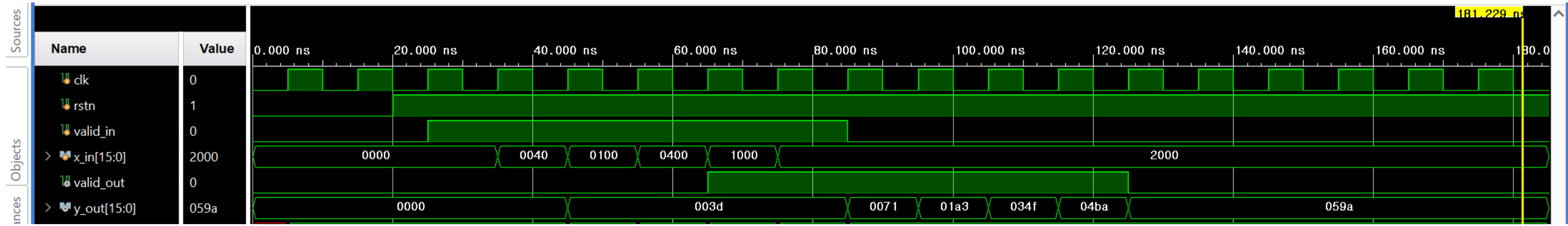
PWL Module Architecture

- yout=slope[i]·xin+intercept[i]

```
71        wire signed [31:0] mul_result;
72        signed_mul mul_unit (
73            .clk(clk),
74            .rst(rst),
75            .a(slope_reg),
76            .b(x_reg),
77            .result(mul_result)
78        );
79
80        wire signed [15:0] mul_shifted = mul_result[23:8];
81        wire signed [15:0] y_out;
82        wire c_out;
83        cla_adder adder_unit (
84            .a(mul_shifted),
85            .b(intercept_reg),
86            .cin(1'b0),
87            .sum(y_temp),
88            .cout(c_out)
89        );
```

LN Approximation Model

PWL Module Architecture

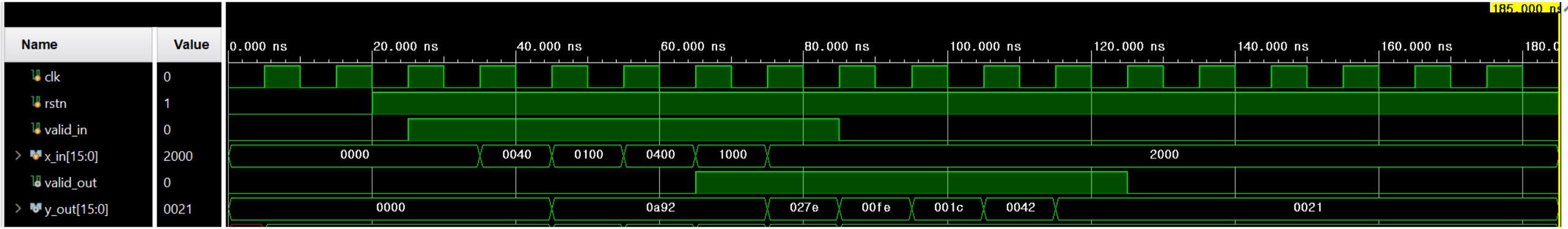


PWL Sqrt Result Comparison

	x_in (float)	expected sqrt(x)	simulated y_out	abs error
1	0.0	0.0	0.23828125	0.23828125
2	0.25	0.5	0.23828125	0.26171875
3	1.0	1.0	0.44140625	0.55859375
4	4.0	2.0	1.63671875	0.36328125
5	16.0	4.0	3.3046875	0.6953125
6	32.0	5.656854249492381	4.7265625	0.9302917494923806

LN Approximation Model

PWL Module Architecture



PWL Reciprocal Sqrt Result

	x_in (float)	expected 1/sqrt(x)	simulated y_out	abs error
1	0.0	10.0	10.5703125	0.5703125
2	0.25	2.0	2.4921875	0.4921875
3	1.0	1.0	0.92578125	0.07421875
4	4.0	0.5	0.26171875	0.23828125
5	16.0	0.25	0.12890625	0.12109375
6	32.0	0.17677669529663687	0.08203125	0.09474544529663687

Compare the Standard and Approximation

Standard LN Model

Operate	IP	latency
DATA Type	**IEEE - 754**	
ADD(SUB)	**Using floating_point_(add)**	**12**
ADDER TREE	Using floating_point_(add)	12 * 6 (6 Level : 64 → 32 .. →1)
MULT	Using floating_point_(mult)	7
MULT 1/N	floating_point_(div)	16
RECIP_SQRT	floating_point_(rsqrt)	5

Approximation LN Model

substituted
Q8.8
CLA
Pairwise
MULT
Bit shift and LUT
Using LUT

Compare the Standard and Approximation

Standard LN Model

Algorithm LayerNormalization(X, γ, β):
Input:
  X ← input vector of length N
  γ, β ← learnable vectors (length N)
Output:
  Y ← normalized and scaled vector

Stage 1: Compute mean μ_x
  $\boldsymbol{\mu_x} \leftarrow (1 / N) \cdot \text{sum\_\{i=0\}\^{}\{N−1\}}\ X_i$
  // using ADDER TREE + DIV IP

Stage 2: Subtract mean (centered input)
  for i ← 0 to N−1 do
    $\mathbf{diff_i} \leftarrow X_i - \mu_x$
  // using SUB IP (can be ADD IP with negated μ_x)

Stage 3: Compute variance σ_x^2 and stddev σ_x
  $\sigma_x^2 \leftarrow (1 / N) \cdot \text{sum\_\{i=0\}\^{}\{N−1\}}\ (\text{diff}_i)^2$
  $\boldsymbol{\sigma_x} \leftarrow \text{sqrt}(\sigma_x^2)$
  // using SQUARE (MULT) IP + ADDER TREE + DIV + RECIP_SQRT IP

Stage 4: Normalize
  for i ← 0 to N−1 do
    $\mathbf{norm_i} \leftarrow \text{diff}_i / \sigma_x$
  // using MULT IP with precomputed $1/\sigma_x$

Stage 5: Scale and Shift
  for i ← 0 to N−1 do
    $\mathbf{Y_i} \leftarrow \gamma_i \cdot \text{norm}_i + \beta_i$
  // using MULT IP + ADD IP

Return Y

Approximation LN Model

Algorithm LayerNormalization(X):
Input:
  X ← input vector of length N (Q8.8 format)
Output:
  Y ← normalized and scaled vector

Stage 1: Compute mean μ_x
  $\boldsymbol{\mu_x} \leftarrow \text{sum\_\{i=0\}\^{}\{N−1\}}\ X_i >> \log_2 n$
  // using ADDER TREE + Bit shift

Stage 2: Pairwise Variance
  (1) Split x into G=16 groups (each 16-dim)
  (2) For each group:
    $\mu_i \leftarrow \text{mean}(x_i)$
    $M_i \leftarrow \Sigma(x_i - \mu_i)^2$
  (3) Pairwise Merge (16→8→4→2→1):
    For every pair:
      $\delta \leftarrow \mu_1 - \mu_2$
      $M_{12} \leftarrow M_1 + M_2 + (n_1 \cdot n_2 \cdot \delta^2) >> \log2(n_1 + n_2)$
      $\mu_{12} \leftarrow (\mu_1 \cdot n_1 + \mu_2 \cdot n_2) >> \log2(n_1 + n_2)$
  (4) Final variance $\sigma^2 \leftarrow$ M_total >> log2(N)
  //using Pairwise + CLA adder + MULT + Bit shift

Stage 3: PWL Approximation
  std_approx ← PWL_SQRT(σ^2)
  recip_std_approx ← PWL_RECIP(std_approx)
  // using LUT + CLA adder + MULT

Stage 4: Normalization (output in float)
  for i ← 0 to N−1 do
  $\mathbf{diff_i} \leftarrow X_i - \mu_x$
  $\mathbf{norm_i} \leftarrow \text{diff}_i \times 1/\sigma_x$
  //using CLA adder + MULT

Plans for Next

- Verilog-based module synthesis and FPGA implementation
- Completion of full LN Approximation module
- Pipelining the LN Approximation process