Midterm Project Report
(Float16 Arithmetic Computation Unit Design)

61275047H 黃聖崴

一、 Float16 Datatype

Float16 Datatype近年來常在AI運算中使用，其優勢為同樣為浮點數的表示法，卻比Float32的Datatype使用更少的位元數，進而可以減少在硬體設計時所消耗的資源，而其劣勢為較少的位元，可以表示的數值範圍與精度(precision)就相對的較低，在計算時精度會比Float32來的差。以下將列出目前在設計浮點數計算時，硬體常用的Datatype：

	Sign bit	Exponent bits	Mantissa bits
BFloat16	1	8	7
Float16	1	5	10
Float32	1	8	23

在原始的設計思維中，我希望透過選擇Float16這個Datatype，達成高精度的運算，並應用在Deep Learning Model Activation Function的加速中，然而因著精度較低的緣故(相較Float32而言)，在計算如Softmax這類的函數，需處理極小值除以極大值的問題，導致於結果失真，但在General case上，以下將介紹的Arithmetic Computation電路都有不錯的效果。

二、 Float16 Adder

在 Float16 Adder 的設計中，由 Float32 加法的啟發，大致可以將 Float16 的加法分為以下步驟：

1. 將輸入的兩個 Float16 數值分別解碼為 sign, exponent, mantissa，方便接下來的計算。

2. 對於兩個數值的 exponent 進行比較，exponent 較小的那個數值將 mantissa 右移，使得兩數的 exponent 對齊(數值相等)。在此步驟，會事先將 mantissa 前面補一個 implicit bit(若為 subnormal 則為 0，否則為 1)，並且在後面補 5 個 0(用於後面 Normalization，會設定 5 個 0 的原因為 Round-to-nearest-even rounding 會需要一個 guard bit、一個 round bit 和多個 bits &(and) 在一起作為 sticky bits)，程式碼如下圖所示。

```
// the part for barrel-shifter, align small number's mantissa to bigger one
assign m1_preshift = (exp_a == 0) ? {1'b0, mant_a, 5'b00000} : {1'b1, mant_a, 5'b00000};
assign m2_preshift = (exp_b == 0) ? {1'b0, mant_b, 5'b00000} : {1'b1, mant_b, 5'b00000};
assign m1_shifted_w = exp_cmp_ab ? m1_preshift : m1_preshift >> exp_diff;
assign m2_shifted_w = exp_cmp_ba ? m2_preshift : m2_preshift >> exp_diff;
```

3. 對於已經位移過的 mantissa 進行運算，若兩數的 sign 相同，則進行相加，否則 mantissa 較大者與 mantissa 較小者進行相減。

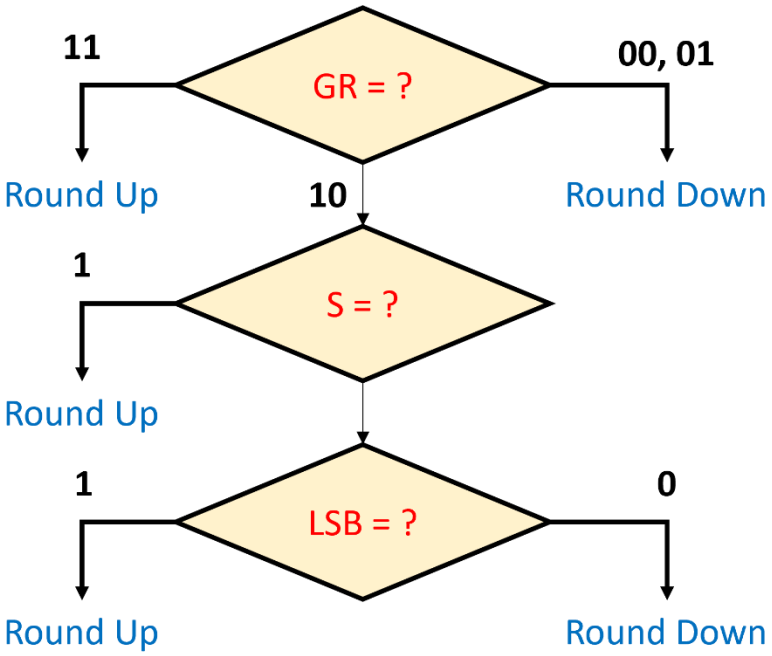
4. 對於剛剛已經運算後的 mantissa，要進行正規化(讓輸出的 mantissa 是以 1.xxxxxxxx 的型態表示)。在此步驟，會透過 priority encoder 去對於 exp_res 和 mant_res 去做修正(程式碼如下所示)。

```
// ========== Third Stage: Mantissa Addition ->  Normal Output ===========

// Normalization
    // 1x.xxxxxxxxxxxxxx -> shift right 1 bit to 01.xxxxxxxxxxxxxx and exp + 1
    // ==> extract [15:1]
    // 01.xxxxxxxxxxxxxx -> maintain mantissa and maintain exponent
    // ==> extract [14:0]
    // 00.1xxxxxxxxxxxxx -> shift left 1 bit to 01.xxxxxxxxxxxxxx and exp - 1
    // ==> extract [13:0] and shift left 1
    // 00.01xxxxxxxxxxxx -> shift left 2 bit to 01.xxxxxxxxxxxxxx and exp - 2
    // ==> extract [12:0] and shift left 2
    // 00.001xxxxxxxxxxx -> shift left 3 bit to 01.xxxxxxxxxxxxxx and exp - 3
    // ==> extract [11:0] and shift left 3
    // 00.0001xxxxxxxxxx -> shift left 4 bit to 01.xxxxxxxxxxxxxx and exp - 4
    // ==> extract [10:0] and shift left 4
    //                      .
    //                      .
    //                      .
    // 00.00000000000001x -> shift left 14 bit to 01.xxxxxxxxxxxxxx and exp - 14
    // ==> extract [0] and shift left 14
```

```
// For Round to nearest, ties to even
always_comb begin
    casez (mant_res_r)
        17'b1????????????????: mant_norm = mant_res_r[15:1];
        17'b01???????????????: mant_norm = mant_res_r[14:0];
        17'b001??????????????: mant_norm = mant_res_r[13:0] << 1;
        17'b0001?????????????: mant_norm = mant_res_r[12:0] << 2;
        17'b00001????????????: mant_norm = mant_res_r[11:0] << 3;
        17'b000001???????????: mant_norm = mant_res_r[10:0] << 4;
        17'b0000001??????????: mant_norm = mant_res_r[9:0] << 5;
        17'b00000001?????????: mant_norm = mant_res_r[8:0] << 6;
        17'b000000001????????: mant_norm = mant_res_r[7:0] << 7;
        17'b0000000001???????: mant_norm = mant_res_r[6:0] << 8;
        17'b00000000001??????: mant_norm = mant_res_r[5:0] << 9;
        17'b000000000001?????: mant_norm = mant_res_r[4:0] << 10;
        17'b0000000000001????: mant_norm = mant_res_r[3:0] << 11;
        17'b00000000000001???: mant_norm = mant_res_r[2:0] << 12;
        17'b000000000000001??: mant_norm = mant_res_r[1:0] << 13;
        17'b0000000000000001?: mant_norm = mant_res_r[0] << 14;
        default: mant_norm = 15'b0;
    endcase
end
```

```
always_comb begin
    casez (mant_res_r)
        17'b1????????????????: exp_norm_tmp = exp_res_sec_r + 1;
        17'b01???????????????: exp_norm_tmp = exp_res_sec_r;
        17'b001??????????????: exp_norm_tmp = exp_res_sec_r - 1;
        17'b0001?????????????: exp_norm_tmp = exp_res_sec_r - 2;
        17'b00001????????????: exp_norm_tmp = exp_res_sec_r - 3;
        17'b000001???????????: exp_norm_tmp = exp_res_sec_r - 4;
        17'b0000001??????????: exp_norm_tmp = exp_res_sec_r - 5;
        17'b00000001?????????: exp_norm_tmp = exp_res_sec_r - 6;
        17'b000000001????????: exp_norm_tmp = exp_res_sec_r - 7;
        17'b0000000001???????: exp_norm_tmp = exp_res_sec_r - 8;
        17'b00000000001??????: exp_norm_tmp = exp_res_sec_r - 9;
        17'b000000000001?????: exp_norm_tmp = exp_res_sec_r - 10;
        17'b0000000000001????: exp_norm_tmp = exp_res_sec_r - 11;
        17'b00000000000001???: exp_norm_tmp = exp_res_sec_r - 12;
        17'b000000000000001??: exp_norm_tmp = exp_res_sec_r - 13;
        17'b0000000000000001?: exp_norm_tmp = exp_res_sec_r - 14;
        default: exp_norm_tmp = 5'b0;
    endcase
end
```

5. 在完成了 Normalization 後，會進行 Round-to-nearest-even rounding，會先取出還沒輸出前的 mantissa(5~14 bits)、lsb(第 5 bits)、guard_bit(第 4 bits)、round_bit(第 3 bits)和 sticky_bit(把 0 ~ 2 bits OR(||)在一起)，並用下圖的 Flow Chart 判斷要不要做 round up，程式碼如下圖。




```
// Rounding: Round-to-Nearest-Even
// Extract GRS(Gound bit, Round bit, Sticky bit)
assign mant_main = mant_norm[14:5];
assign lsb = mant_norm[5];
assign guard_bit = mant_norm[4];
assign round_bit = mant_norm[3];
assign sticky_bit = | mant_norm[2:0];

// Round-to-Nearest-Even judge
// GR = 11 or GRS = 101 or LGRS = 1100
assign round_up = ((guard_bit && round_bit) || (guard_bit && ~round_bit && sticky_bit) || (lsb && guard_bit && ~round_bit && ~sticky_bit));
```

6. 最後將 round up 加到 normalization 後的 mantissa 即可。

以下為 Float16 Adder 的 simulation result、utilization 以及 timing analysis 的結果：

Simulation Result


```
PASS at 995
  a      = -91.542000 (0x51f2)
  b      = 35.476000 (0x5619)
  DUT    = 0xd303 (-56.093750)
  GOLDEN = -56.066000
  ERROR = 0.027750
PASS at 996
  a      = 10.002000 (0x5145)
  b      = -79.655000 (0x54b7)
  DUT    = 0xd45a (-69.625000)
  GOLDEN = -69.653000
  ERROR = 0.028000
PASS at 997
  a      = 47.547000 (0xc4cd)
  b      = 97.566000 (0xd4ff)
  DUT    = 0x5889 (145.125000)
  GOLDEN = 145.113000
  ERROR = 0.012000
PASS at 998
  a      = 42.157000 (0xc4cd)
  b      = 75.446000 (0xd4ff)
  DUT    = 0x575a (117.625000)
  GOLDEN = 117.603000
  ERROR = 0.022000
PASS at 999
  a      = -4.800000 (0xc4cd)
  b      = -79.957000 (0xd4ff)
  DUT    = 0xd54c (-84.750000)
  GOLDEN = -84.757000
  ERROR = 0.007000

Test Finished. Total: 1000. Failures: 0
```

上圖為 testbench 在 tcl console 上 auto compare 的結果，可以觀察到，在設定硬體計算數值與真實數值相減相差< 0.2 模擬結果為 PASS 的情況下，1000 個 random pattern 皆為 PASS，testbench 程式碼如下圖所示。

```
error = fp16_to_float(dut_out) - golden_result[ch_i - 2];
error_abs = (error > 0) ? error : -error;

if(error_abs > 0.2) begin
    $display("MISMATCH at %0d", ch_i - 2);
    $display("  a      = %f (0x%h)", ra[ch_i - 2], a_in);
    $display("  b      = %f (0x%h)", rb[ch_i - 2], b_in);
    $display("  DUT    = 0x%h (%.6f)", dut_out, fp16_to_float(dut_out));
    $display("  GOLDEN = %.6f", golden_result[ch_i - 2]);
    $display("  ERROR = %.6f", error_abs);
    fail_count++;
end else begin
    $display("PASS at %0d", ch_i - 2);
    $display("  a      = %f (0x%h)", ra[ch_i - 2], a_in);
    $display("  b      = %f (0x%h)", rb[ch_i - 2], b_in);
    $display("  DUT    = 0x%h (%.6f)", dut_out, fp16_to_float(dut_out));
    $display("  GOLDEN = %.6f", golden_result[ch_i- 2]);
    $display("  ERROR = %.6f", error_abs);
end
```

Utilization

Resource	Estimation	Available	Utilization %
LUT	287	871680	0.03
FF	103	1743360	0.01
IO	50	416	12.02
BUFG	1	672	0.15

Timing Report

Design Timing Summary

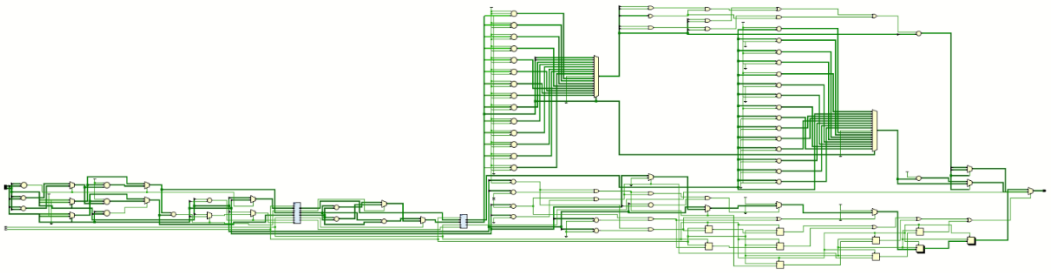
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.948 ns	Worst Hold Slack (WHS):	-0.057 ns	Worst Pulse Width Slack (WPWS):	4.725 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	-2.204 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	43	Number of Failing Endpoints:	0
Total Number of Endpoints:	43	Total Number of Endpoints:	43	Total Number of Endpoints:	104

Timing constraints are not met.

在 Timing Analysis 部分，由於我在 Float16 Adder 中切了兩級的 pipeline(分別在 barrel-shifter(exponent alignment)和 mantissa addition 後面)，故在 Setup Time 上有不錯的表現。

我使用的 clock rate 為 100MHz(clock period 為 10.000ns)，可以看出還有 8.948ns 的 Setup Time Slack，代表 Float16 Adder 可以跑到更快的 clock rate，Hold Time Violation 部分可忽略(要 implement 時再解即可)。

Elaborated Design Schematic



三、 Log scale multiplier

由於 Float16 在 precision 方面(mantissa bits 數)相較於 Float32 沒有那麼大的餘裕，若用傳統的浮點數乘法計算，會在 normalization 時被限制可以校正的空間。

因此，我想到若將乘法轉換到 logarithm domain，乘法運算可以被轉換成加法運算，並在要輸出前轉換為 2 的冪次方即可，如以下數值式：

$$a \times b = 2^{\log_2(a \times b)} = 2^{\log_2 a + \log_2 b}$$

透過上述的數學方法，我將其實做成電路，並將$\log_2 x$的計算與2^x的運算透過查找表來完成，以下我將進一步介紹硬體設計的想法：

一、 先將$\log_2 x$和2^x計算所需的查找表 load 進 design，提供後續計算使用。在$\log_2 x$和2^x的查找表部分，我使用 python code 分別產生出$\log_2 x$運算所需的 mantissa 和2^x運算所需反查的完整 Float16 數值，都有 128 筆，程式碼如下圖所示：

```python
import math, struct

# log2 LUT
with open('log2_lut_128_new.txt', 'w') as f:
    for i in range(128):
        m = (i << 3) / 1024.0                # 10-bit mantissa前 7 bits
        val = int(round((math.log2(1+m)) * 1024))
        f.write(f"{val:03x}\n")              # 10 位元→3 字 hex
# exp2 LUT
def fp16(v):
    return struct.unpack(">H", struct.pack(">e", v))[0]
with open('exp2_lut_128_new.txt', 'w') as f:
    for i in range(128):
        frac = i / 128.0
        f.write(f"{fp16(2**frac):04x}\n")
```

```
// ========== Look Up Table Initialization ===========
// procedure block for handle lut_wr_ptr
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        lut_wr_ptr <= 8'b0;
    end else begin
        lut_wr_ptr <= (lut_wr_en) ? lut_wr_ptr + 1 : lut_wr_ptr;
    end
end

// procedure block for handle log2_lut0
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        log2_lut0[lut_wr_ptr] <= 16'b0;
    end else begin
        log2_lut0[lut_wr_ptr] <= (lut_wr_en) ? log2_lut_data_in : log2_lut0[lut_wr_ptr];
    end
end

// procedure block for handle log2_lut1
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        log2_lut1[lut_wr_ptr] <= 16'b0;
    end else begin
        log2_lut1[lut_wr_ptr] <= (lut_wr_en) ? log2_lut_data_in : log2_lut1[lut_wr_ptr];
    end
end

// procedure block for handle exp2_lut
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        exp2_lut[lut_wr_ptr] <= 16'b0;
    end else begin
        exp2_lut[lut_wr_ptr] <= (lut_wr_en) ? exp2_lut_data_in : exp2_lut[lut_wr_ptr];
    end
end
```

\

二、 接著，如同 Float16 Adder 一樣將輸入的兩個 Float16 數值分別解碼為 sign, exponent, mantissa，方便接下來的計算，並將 exponent 做 unbias 的動作(減去 15)。

三、

```
// ========== First Stage: Unpack float16 a and b ===========

// separate each part of input
assign {sign_a_fir_w, exp_a_raw, mant_a} = a;
assign {sign_b_fir_w, exp_b_raw, mant_b} = b;

// get the read address of log2_lut0 & log2_lut0
assign log2_idx_a = mant_a[9:3];
assign log2_idx_b = mant_b[9:3];

// get the unbiased exponent
assign exp_a_fir_w = $signed({2'b00, exp_a_raw}) - 7'sd15;
assign exp_b_fir_w = $signed({2'b00, exp_b_raw}) - 7'sd15;
```

四、 接下來將 mantissa 的前 7 bits 作為 look up $\log_2 x$的 idx，可得到$\log_2 x$的 mantissa 值。

```
// procedure block for get the log2-mantissa of input a
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        log2_mant_a <= 10'b0;
    end else begin
        // log2_mant_a <= (lut_wr_done) ? log2_lut0[log2_idx_a] : 10'b0;
        log2_mant_a <= log2_lut0[log2_idx_a];
    end
end

// procedure block for get the log2-mantissa of input b
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        log2_mant_b <= 10'b0;
    end else begin
        // log2_mant_b <= (lut_wr_done) ? log2_lut1[log2_idx_b] : 10'b0;
        log2_mant_b <= log2_lut1[log2_idx_b];
    end
end
```

五、 接著將 unbiased 的 exponent 和 mantissa 做相加，並對於相加結果 normalize 後，將這時的 mantissa 值用於查找 2^x 的表還原真實數值。

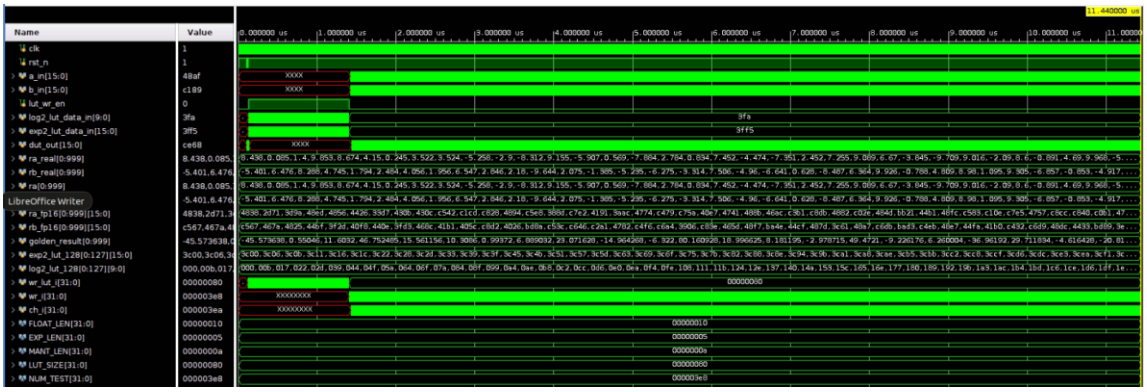
```
// ========== Second Stage: Compute exponent/mantissa sum and normalize ==========

// pipeline for sign_a & sign_b
assign sign_a_sec_w = sign_a_fir_r;
assign sign_b_sec_w = sign_b_fir_r;

// compute exponent/mantissa sum
assign exp_sum = exp_a_fir_r + exp_b_fir_r;
assign mant_sum = {2'b00, log2_mant_a} + {2'b00, log2_mant_b};

// normalize the sum of exponent/mantissa
always_comb begin
    if(mant_sum[10]) begin
        exp_norm_w = exp_sum + 7'sd1;
    end else begin
        exp_norm_w = exp_sum;
    end
end

assign mant_norm = mant_sum[9:0];

// get the index of exp2 look up
assign exp2_idx = mant_norm[9:3];
```

六、 最後，將 bias 的值加回來，並對於要 concatenate 的結果在進行一次輸出的分析。

```
// ========== Third Stage: Exponent adjust & concatenate ===========

// get the sign of the final number
assign sign_final = sign_a_sec_r ^ sign_b_sec_r;

// get the exponent of the final number
assign exp_final = exp_norm_r + 7'sd15;

// get the mantissa of the final number
assign mant_final = exp2_val[9:0];

always_comb begin
    if(exp_final <= 0 && exp_norm_r >= -MANT_LEN) begin
        // subnormal
        subnormal_imp = {1'b1, mant_final};
        subnormal_shifted = subnormal_imp >> (1 - exp_final);
        normal_result = {sign_final, 5'h00, subnormal_shifted};
    end else if(exp_final <= 0) begin
        // underflow to zero
        normal_result = {sign_final, 5'h00, 10'h000};
    end else if(exp_final >= 30) begin
        // overflow to inf
        normal_result = {sign_final, 5'h1F, 10'h000};
    end else begin
        // normal
        normal_result = {sign_final, exp_final[4:0], mant_final};
    end
end
```

以下為 Float16 Adder 的 simulation result、utilization 以及 timing analysis 的結果：

Simulation Result



上圖為 simulation waveform，可以觀察到 rst_n realease 後的 128 個 cycle 在做 look up table 的 input，後面完成 1000 個 pattern 的運算。

```
PASS at 996
  a      = -4.038000 (0x3266)
  b      = -2.888000 (0x4838)
  DUT    = 0x49c8 (11.562500)
  GOLDEN = 11.661744
  ERROR = 0.099244
  ERROR RATE = 0.008510
PASS at 997
  a      = -5.201000 (0x48af)
  b      = 8.661000 (0xc189)
  DUT    = 0xd191 (-44.531250)
  GOLDEN = -45.045861
  ERROR = 0.514611
  ERROR RATE = 0.011424
PASS at 998
  a      = 0.200000 (0x48af)
  b      = 8.441000 (0xc189)
  DUT    = 0x3eba (1.681641)
  GOLDEN = 1.688200
  ERROR = 0.006559
  ERROR RATE = 0.003885
PASS at 999
  a      = 9.370000 (0x48af)
  b      = -2.768000 (0xc189)
  DUT    = 0xce68 (-25.625000)
  GOLDEN = -25.936160
  ERROR = 0.311160
  ERROR RATE = 0.011997

 Test Finished. Total: 1000, Failures: 0
```

上圖為 testbench 在 tcl console 上 auto compare 的結果，可以觀察到，在設定誤差值(硬體輸出與實際輸出差值的絕對值)與真實數值絕對值所組成的 error rate < 1.9%時，模擬結果為 PASS 的情況下，1000 個 random pattern 皆為 PASS，testbench 程式碼如下圖所示。

```
error = fp16_to_float(dut_out) - golden_result[ch_i - 2];
error_abs = (error > 0) ? error : -error;
golden_abs = (golden_result[ch_i - 2] > 0) ? golden_result[ch_i - 2] : -golden_result[ch_i - 2];
error_rate = error_abs/golden_abs;

if(error_abs > 0.019 * golden_abs) begin
    $display("MISMATCH at %0d", ch_i - 2);
    $display("  a       = %f (0x%h)", ra[ch_i - 2], a_in);
    $display("  b       = %f (0x%h)", rb[ch_i - 2], b_in);
    $display("  DUT     = 0x%h (%.6f)", dut_out, fp16_to_float(dut_out));
    $display("  GOLDEN = %.6f", golden_result[ch_i -2]);
    $display("  ERROR = %.6f", error_abs);
    $display("  ERROR RATE = %.6f", error_rate);
    fail_count++;
end else begin
    $display("PASS at %0d", ch_i - 2);
    $display("  a       = %f (0x%h)", ra[ch_i - 2], a_in);
    $display("  b       = %f (0x%h)", rb[ch_i - 2], b_in);
    $display("  DUT     = 0x%h (%.6f)", dut_out, fp16_to_float(dut_out));
    $display("  GOLDEN = %.6f", golden_result[ch_i- 2]);
    $display("  ERROR = %.6f", error_abs);
    $display("  ERROR RATE = %.6f", error_rate);
end
```

Utilization

Resource	Estimation	Available	Utilization %
LUT	172	871680	0.02
LUTRAM	80	403200	0.02
FF	72	1743360	0.01
IO	75	416	18.03
BUFG	1	672	0.15

LUTRAM 為 Look Up Table 所使用的 resource

Timing Report

Design Timing Summary

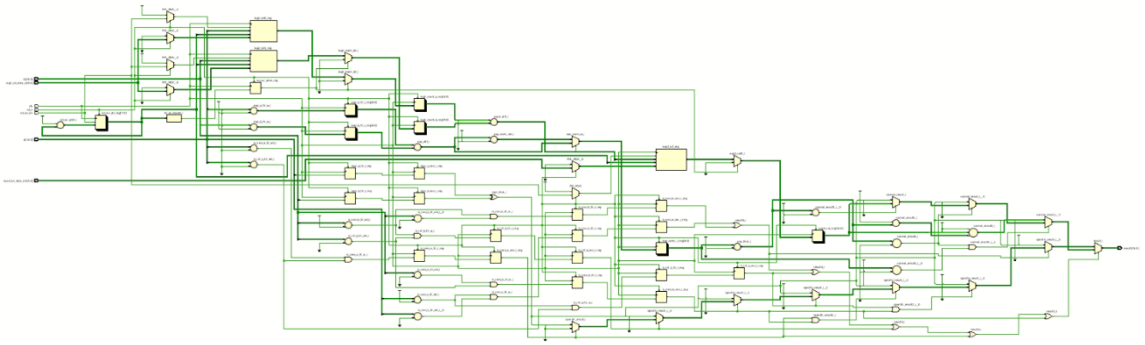
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.545 ns	Worst Hold Slack (WHS):	-0.078 ns	Worst Pulse Width Slack (WPWS):	4.468 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	-30.214 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	502	Number of Failing Endpoints:	0
Total Number of Endpoints:	612	Total Number of Endpoints:	612	Total Number of Endpoints:	153

Timing constraints are not met.

在 Timing Analysis 部分，由於我在 Log scale multiplier 中也切了兩級的 pipeline(分別在 look up $\log_2 x$和2^x的 table 處)，故在 Setup Time 上有不錯的表現。

我使用的 clock rate 為 100MHz(clock period 為 10.000ns)，可以看出還有 8.545ns 的 Setup Time Slack，代表 Float16 Adder 可以跑到更快的 clock rate，Hold Time Violation 部分可忽略(要 implement 時再解即可)。

Elaborated Design Schematic



四、 Log scale divider

與Log scale multiplier相同的想法，由於Float16在precision方面(mantissa bits數)相較於Float32沒有那麼大的餘裕，若用傳統的浮點數乘法計算，會在normalization時被限制可以校正的空間。

相同地，我想到若將除法轉換到logarithm domain，除法運算可以被轉換成減法運算，並在要輸出前轉換為2的冪次方即可，如以下數值式：

$$a/b = 2^{\log_2(a/b)} = 2^{\log_2 a - \log_2 b}$$

透過上述的數學方法，我將其實做成電路，並將$\log_2 x$的計算與2^x的運算透過查找表來完成，以下我將進一步介紹硬體設計的想法：

i. 先將$\log_2 x$和2^x計算所需的查找表 load 進 design，提供後續計算使用。在$\log_2 x$和2^x的查找表部分，我使用 python code 分別產生出$\log_2 x$運算所需的 mantissa 和2^x運算所需反查的完整 Float16 數值，都有 128 筆，程式碼如下圖所示：

```python
import math, struct

# log2 LUT
with open('log2_lut_128_new.txt', 'w') as f:
    for i in range(128):
        m = (i << 3) / 1024.0                # 10-bit mantissa前 7 bits
        val = int(round((math.log2(1+m)) * 1024))
        f.write(f"{val:03x}\n")              # 10 位元→3 字 hex
# exp2 LUT
def fp16(v):
    return struct.unpack(">H", struct.pack(">e", v))[0]
with open('exp2_lut_128_new.txt', 'w') as f:
    for i in range(128):
        frac = i / 128.0
        f.write(f"{fp16(2**frac):04x}\n")
```

```systemverilog
// ========== Look Up Table Initialization ===========
// procedure block for handle lut_wr_ptr
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        lut_wr_ptr <= 8'b0;
    end else begin
        lut_wr_ptr <= (lut_wr_en) ? lut_wr_ptr + 1 : lut_wr_ptr;
    end
end

// procedure block for handle log2_lut0
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        log2_lut0[lut_wr_ptr] <= 16'b0;
    end else begin
        log2_lut0[lut_wr_ptr] <= (lut_wr_en) ? log2_lut_data_in : log2_lut0[lut_wr_ptr];
    end
end

// procedure block for handle log2_lut1
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        log2_lut1[lut_wr_ptr] <= 16'b0;
    end else begin
        log2_lut1[lut_wr_ptr] <= (lut_wr_en) ? log2_lut_data_in : log2_lut1[lut_wr_ptr];
    end
end

// procedure block for handle exp2_lut
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        exp2_lut[lut_wr_ptr] <= 16'b0;
    end else begin
        exp2_lut[lut_wr_ptr] <= (lut_wr_en) ? exp2_lut_data_in : exp2_lut[lut_wr_ptr];
    end
end
```

ii. 接著，如同Log scale multiplier一樣將輸入的兩個Float16數值分別解碼為sign, exponent, mantissa，方便接下來的計算，並將exponent做unbias的動作(減去15)。

```
// ========== First Stage: Unpack float16 a and b ===========

// separate each part of input
assign {sign_a_fir_w, exp_a_raw, mant_a} = a;
assign {sign_b_fir_w, exp_b_raw, mant_b} = b;

// get the read address of log2_lut0 & log2_lut0
assign log2_idx_a = mant_a[9:3];
assign log2_idx_b = mant_b[9:3];

// get the unbiased exponent
assign exp_a_fir_w = $signed({2'b00, exp_a_raw}) - 7'sd15;
assign exp_b_fir_w = $signed({2'b00, exp_b_raw}) - 7'sd15;
```

iii. 接下來將mantissa的前7 bits作為look up $\log_2 x$的idx，可得到$\log_2 x$的mantissa值。

```
// procedure block for get the log2-mantissa of input a
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        log2_mant_a <= 10'b0;
    end else begin
        // log2_mant_a <= (lut_wr_done) ? log2_lut0[log2_idx_a] : 10'b0;
        log2_mant_a <= log2_lut0[log2_idx_a];
    end
end

// procedure block for get the log2-mantissa of input b
always_ff @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        log2_mant_b <= 10'b0;
    end else begin
        // log2_mant_b <= (lut_wr_done) ? log2_lut1[log2_idx_b] : 10'b0;
        log2_mant_b <= log2_lut1[log2_idx_b];
    end
end
```

iv. 異於乘法的是，除法將unbiased的exponent和mantissa做相減，並對於相減結果normalize後(exp_diff - 1)，將這時的mantissa值用於查找2^x的表還原真實數值。

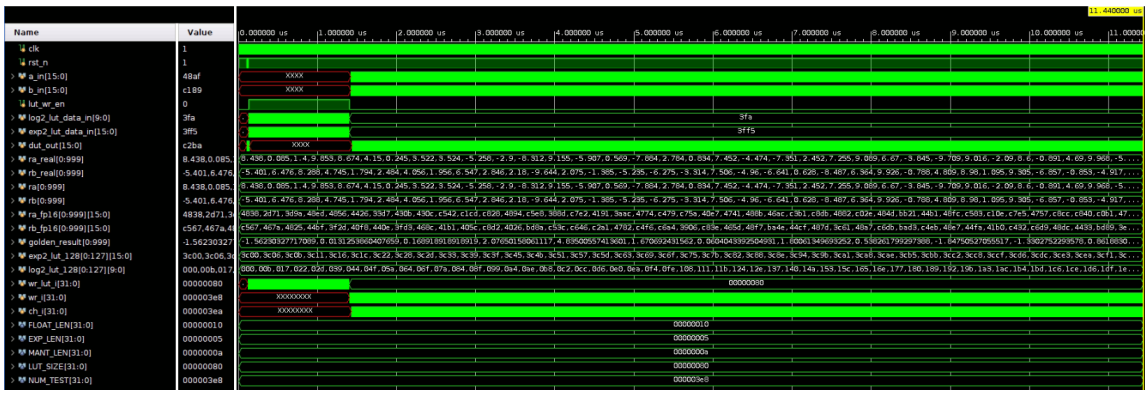
```
// ========== Second Stage: Compute exponent/mantissa sum and normalize ===========

// pipeline for sign_a & sign_b
assign sign_a_sec_w = sign_a_fir_r;
assign sign_b_sec_w = sign_b_fir_r;

// compute exponent/mantissa sum
assign exp_diff = exp_a_fir_r - exp_b_fir_r;
assign mant_diff = {2'b00, log2_mant_a} - {2'b00, log2_mant_b};

// normalize the sum of exponent/mantissa
always_comb begin
    if(mant_diff[11]) begin
        exp_norm_w = exp_diff - 7'sd1;
    end else begin
        exp_norm_w = exp_diff;
    end
end

assign mant_norm = mant_diff[9:0];

// get the index of exp2 look up
assign exp2_idx = mant_norm[9:3];
```

v. 最後，將bias的值加回來，並對於要concatenate的結果在進行一次輸出的分析。

```
// ========== Third Stage: Exponent adjust & concatenate ===========

// get the sign of the final number
assign sign_final = sign_a_sec_r ^ sign_b_sec_r;

// get the exponent of the final number
assign exp_final = exp_norm_r + 7'sd15;

// get the mantissa of the final number
assign mant_final = exp2_val[9:0];

// procedure block for normal result
always_comb begin
    if(exp_final <= 0 && exp_norm_r >= -MANT_LEN) begin
        // subnormal
        subnormal_imp = {1'b1, mant_final};
        subnormal_shifted = subnormal_imp >> (1 - exp_final);
        normal_result = {sign_final, 5'h00, subnormal_shifted};
    end else if(exp_final <= 0) begin
        // underflow to zero
        normal_result = {sign_final, 5'h00, 10'h000};
    end else if(exp_final > 30) begin
        // overflow to inf
        normal_result = {sign_final, 5'h1F, 10'h000};
    end else begin
        // normal
        normal_result = {sign_final, exp_final[4:0], mant_final};
    end
end
```

以下為 Float16 Adder 的 simulation result、utilization 以及 timing analysis 的結果：

Simulation Result



上圖為 simulation waveform，同 Log scale multiplier 可以觀察到，rst_n realease 後的 128 個 cycle 在做 look up table 的 input，後面完成 1000 個 pattern 的運算。

```
PASS at 996
  a      = -4.038000 (0x3266)
  b      = -2.888000 (0x4838)
  DUT    = 0x3d99 (1.399414)
  GOLDEN = 1.398199
  ERROR = 0.001215
  ERROR RATE = 0.000869
PASS at 997
  a      = -5.201000 (0x48af)
  b      = 8.661000 (0xc189)
  DUT    = 0xb8cf (-0.601074)
  GOLDEN = -0.600508
  ERROR = 0.000566
  ERROR RATE = 0.000943
PASS at 998
  a      = 0.200000 (0x48af)
  b      = 8.441000 (0xc189)
  DUT    = 0x2609 (0.023575)
  GOLDEN = 0.023694
  ERROR = 0.000119
  ERROR RATE = 0.005024
PASS at 999
  a      = 9.370000 (0x48af)
  b      = -2.768000 (0xc189)
  DUT    = 0xc2ba (-3.363281)
  GOLDEN = -3.385116
  ERROR = 0.021834
  ERROR RATE = 0.006450

Test Finished. Total: 1000, Failures: 0
```

上圖為 testbench 在 tcl console 上 auto compare 的結果，可以觀察到，在設定誤差值(硬體輸出與實際輸出差值的絕對值)與真實數值絕對值所組成的 error rate < 1.1%時，模擬結果為 PASS 的情況下，1000 個 random pattern 皆為 PASS，testbench 程式碼如下圖所示。

Utilization

Resource	Estimation	Available	Utilization %
LUT	170	871680	0.02
LUTRAM	80	403200	0.02
FF	72	1743360	0.01
IO	75	416	18.03
BUFG	1	672	0.15

LUTRAM 為 Look Up Table 所使用的 resource

Timing Report

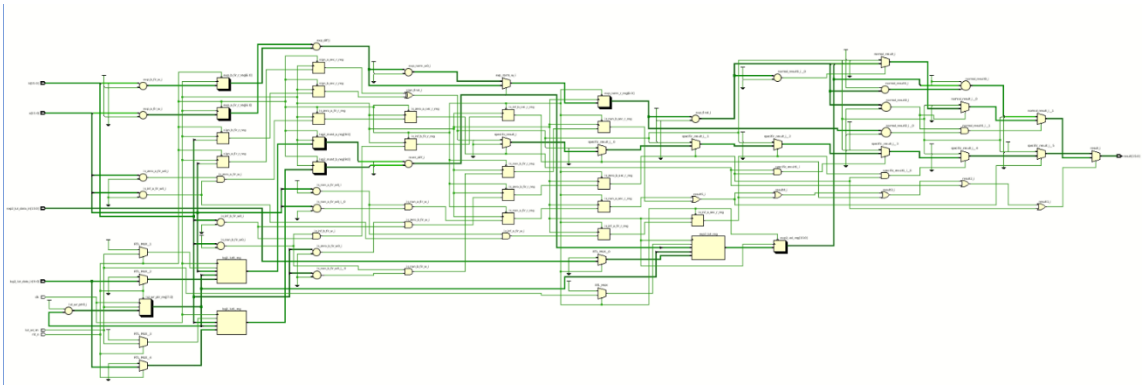
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.729 ns	Worst Hold Slack (WHS):	-0.099 ns	Worst Pulse Width Slack (WPWS):	4.468 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	-30.365 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	502	Number of Failing Endpoints:	0
Total Number of Endpoints:	612	Total Number of Endpoints:	612	Total Number of Endpoints:	153

Timing constraints are not met.

在 Timing Analysis 部分，由於我在 Log scale multiplier 中也切了兩級的 pipeline(分別在 look up $\log_2 x$和2^x的 table 處)，故在 Setup Time 上有不錯的表現。
我使用的 clock rate 為 100MHz(clock period 為 10.000ns)，可以看出還有 8.729ns 的 Setup Time Slack，代表 Float16 Adder 可以跑到更快的 clock rate，Hold Time Violation 部分可忽略(要 implement 時再解即可)。

Elaborated Design Schematic



五、 GitHub Link

上述的電路設計程式碼均可在以下網址查看：
https://github.com/kevin33713371/2025_SPRING_NTNU_DSP_Architecture_Design