首先梳理一下mac模块的功能：
exp　做乘法的阶码求和，然后比较选出最大阶码和的值，计算每个阶码和与最大值的差值exp_sft，送入mul中做移位。
nan　处理nan，用了树形的二分结构，不断的二分，检查是否存在nan，从LSB开始，找到第一个NAN输出尾数out_nan_mts和标志位。
mul　处理乘法，使用booth乘法器，通过CSA树压缩部分积为sum和carry，根据exp_sft移位为非规格化数输出。
mac　处理64对乘加操作，支持INT8、INT16、FP16的普通计算和Winograd。

下面是一些比较难看懂的技巧性的处理解析：

mul模块

1. booth乘法器中，部分积的MSB取符号位的反。
解读：这么做的目的是将正或负的部分积统一为正数，方便扩展位宽时上0就行，不用对负数补符号位1。**这么做的后果是每个部分积需要减去17'h10000，才是正确的值！**
对于整个mul模块，8个部分积，做对齐之后，非INT8需要减去32'h5555_0000才是正确的值。对于INT8，相应的缺口是16'h5500.

```
712     case({is_8bit, in_code})
713         ///////// for 16bit ////////
714         // +/- 0*src_data
715         4'b0000,
716         4'b0111:
717         begin
718             out_data = 17'h10000;
719             out_inv = 1'b0;
720         end
721
722         // + 1*src_data
723         4'b0001,
724         4'b0010:
725         begin
726             out_data = {~src_data[15], src_data};
727             out_inv = 1'b0;
728         end
```

2. 部分积送入CSA之前进行对齐，对齐的同时将Booth编码中减法（取反加一只做了取反）缺的+1补上了，Booth中的out_inv表征是否加1。

```
469     ppre_0 = {7'b0, sel_data_0};
470     ppre_1 = {5'b0, sel_data_1, 1'b0, sel_inv_0};
471     ppre_2 = {3'b0, sel_data_2, 1'b0, sel_inv_1, 2'b0};
472     ppre_3 = {1'b0, sel_data_3, 1'b0, sel_inv_2, 4'b0};
473     ppre_4 = {17'b0, sel_inv_3, 6'b0};
```

3. shift逻辑中， exp_sft[3]为1时将会移位32位以上，移位完就只剩0了，所以单独拎出来判断了。exp_sft的实际值是{exp_sft，2'b00}，所以移位操作移位这么多。

```
588     pp_fp16_0_sft[31:0] = (~cfg_is_fp16_d1[2] | exp_sft[3]) ? 32'b0 :
589                           (pp_out_l1n0_0 >> {exp_sft[2:0], 2'b0});
```

4. pp_sign_tag的含义？

```
606     pp_sign_tag = (~op_out_pvld[1] | exp_sft[3]) ? 8'h0 :
607                   (8'hf0 >> exp_sft[2:0]);
```

对于FP16，为了对齐最大阶码，结果右移{exp_sft，2'b00}。因为Booth编码不是标准的，整个乘法需要减去**32'h5555_0000才是正确的值。**

这个值也应该相应右移{exp_sft，2'b00}位，简单枚举一下：32'h5555_0000 >> {exp_sft[2:0], 2'b0}

```
1  exp_sft = 3'd0  -->  32'h5555_0000
2  exp_sft = 3'd1  -->  32'h0555_5000
3  exp_sft = 3'd2  -->  32'h0055_5500
4  exp_sft = 3'd3  -->  32'h0005_5550
5         ...
6  exp_sft = 3'd7  -->  32'h0000_0005
```

可以看出这个形式和 8'hf0 >> exp_sft[2:0] 很像，将32'h5555_0000简化为 8'hf0 来指代，**每个符号1'b1实际代表4'h5。**

对于FP16来说，这个pp_sign_tag将决定后续需要减去的数，在mac里会有大量的操作。

5. res_gate的含义，这个就是前面说到的需要减去的数。

```
619     res_a_gate = cfg_is_int8_d1 ? 32'h55005500 : //chenhao: int8
620                  ~cfg_is_fp16_d1[3] ? 32'h55550000 : //chenhao: int16
621                  32'h0; //chenhao: fp16
622     res_b_gate = 32'h0;
```

mac模块

梳理功能：

数据位：输入64个mul得到的sum和carry，共64x2个。通过第一级的CSA树4->1，得到16x2；将16x2组合成8x4，CSA2->1得8x2 。此处开始分叉为winograd操作和普通加法。通过多路选择器执行分叉选择。选择winograd操作将执行下述计算的左乘，分为first[1,1,1,0]和second[0,1,-1,-1]，结果将是2x4x2个。重新组合成4x4，CSA压缩为2x4。

```
                  | pp_out_00, pp_out_01, pp_out_02, pp_out_03 |   | 1   0 |
| 1,  1,  1,  0 |  | pp_out_04, pp_out_05, pp_out_06, pp_out_07 |   | 1,  1 |
| 0,  1, -1, -1 |  | pp_out_08, pp_out_09, pp_out_10, pp_out_11 |   | 1, -1 |
                  | pp_out_12, pp_out_13, pp_out_14, pp_out_15 |   | 0, -1 |
```

tag位：输入是64个8bit的pp_sign_tag，按位分组为b[0-7][63:0]，4->1压缩为8x16（b[0-7] sum[0-15]），然后按照winograd的运算按照下述排列计算8个相应的n0,n1,n2,n3，得8x4项。组合成16x2，将1还原成5，得16x4项，进CSA压缩到16x2；将n[0-3]的b[0-7]对齐，得到4x8项，压缩得4x2；这就对应着winograd的n0-n3.

```
                  | 0  4   8  12 |   | 1   0 |
| 1,  1,  1,  0 |  | 1  5   9  13 |   | 1,  1 | —— | n0  n2 |
| 0,  1, -1, -1 |  | 2  6  10  14 |   | 1, -1 | —— | n1  n3 |
                  | 3  7  11  15 |   | 0, -1 |
```

前面数据位只做了左乘，继续做右乘，并加上tag位运算得到的n0-n3，得4x2项，算得补上了tag的n0-n3。最后补上对应的常数pp_sign_patch_[0-3]就得到了输出。

对于非WG算法来说，n0部分就是结果，n1/n2/n3都是0.

pp_sign_patch

mac代码里比较难懂的地方在于tag树和最后的pp_sign_patch如何补全非标准的mul得到正确的结果？下面从pp_sign_patch的计算来说明代码中的一些技巧处理：

首先明确一下结果的位数，64个数相加，结果扩展6位，对于INT8来说，有效位为22位，对INT16或者FP16来说是38位。下面计算patch。

INT8和INT16

对INT8而言，在booth的mul中，每一次乘法计算的结果需要减去16'h5500才得到正确结果。因此加法树的结果与正确结果之间的关系为：sum=tree-64*16'h5500;

patch就是-64*16'h5500，计算一下这个数的补码表示，取反加一为：

```
1   INT8: patch_0 = -64*16'h5500 = -22'b01_0101_0100_0000_0000_0000 =
    22'sb10_1010_1100_0000_0000_0000=22'h2ac000 一致
```

与代码里的patch吻合！

同理对于INT16的patch：

```
INT16: patch_0 = -64*32'h5555_0000=-37'h15_5540_0000=38'sh2a_aac0_0000 一致
```

对于wg的情况下，怎么算？有两个地方不是标准的运算：

1. mul涉及的缺，wg的结果2*2矩阵，各自的patch由所涉及的个数决定，减去相应个数的16'h5500 / 32'h5555_0000。

2. 在wg的计算中，存在减法。

前半部分做减法的时候，如8250行，取反&mask2_4，对INT8来说，有效位22位，此时只有19位，高两位取反应该是11，被mask强制为0，缺口是-20'h8_0000+1；对INT16来说，有效位为38位，此时已有38位做取反，不存在符号位的缺口，只缺+1；

后半部分做减法时，10124行，取反&mask4_2，对INT8，mask4_2是22位；对INT16是46位，都有足够的位数，缺口是+1；

```
8247    mask2_4 = {2'b0, {2{cfg_is_int8_d1[8]}}, 17'h1ffff, {2{~cfg_is_int8_d1[8]}}, 19'h7ffff};
8248    pp_in_l2n4_0 = ~cfg_is_wg_d1[4] ? 42'b0 : pp_in_l1n0_2;
8249    pp_in_l2n4_1 = ~cfg_is_wg_d1[4] ? 42'b0 : pp_in_l1n0_3;
8250    pp_in_l2n4_2 = ~cfg_is_wg_d1[4] ? 42'b0 : ~pp_out_l1n1_0 & mask2_4;
8251    pp_in_l2n4_3 = ~cfg_is_wg_d1[4] ? 42'b0 : ~pp_out_l1n1_1 & mask2_4;
```

```
10121    mask4_2 = {22'h3fffff, {2{~cfg_is_int8_d2[4]}}, 22'h3fffff};
10122    pp_in_l4n2_0 = ~cfg_is_wg_d2[2] ? 46'b0 : pp_in_l3n0_2;
10123    pp_in_l4n2_1 = ~cfg_is_wg_d2[2] ? 46'b0 : pp_in_l3n0_3;
10124    pp_in_l4n2_2 = ~cfg_is_wg_d2[2] ? 46'b0 : ~pp_out_l3n1_0 & mask4_2;
10125    pp_in_l4n2_3 = ~cfg_is_wg_d2[2] ? 46'b0 : ~pp_out_l3n1_1 & mask4_2;
10126    pp_in_l4n2_4 = (~cfg_is_wg_d2[2] | ~cfg_is_fp16_d2[6]) ? 46'b0 : {8'b0, ps_out_l2n2_0};
10127    pp_in_l4n2_5 = (~cfg_is_wg_d2[2] | ~cfg_is_fp16_d2[6]) ? 46'b0 : {8'b0, ps_out_l2n2_1};
```

对1的缺口：从wg的矩阵计算形式（或者从结果往上顺藤摸瓜梳理），可以得到 n0 对应了 4x4 矩阵中的左上角的9个，涉及 9/16*64 个乘积。n1 n2 n3分析类似，比例分别为-3/16 -3/16 1/16。

对2的缺口：需要补的1的个数由矩阵运算中乘以-1的次数决定，n1有6次取反，n2有2次取反，n3有+2-(+2)+1-(+2)+1 = 0次，恰好对抵掉了。

由此可以求出 WG_INT16 和 WG_INT8 的 patch 为：(**在计算结果后面标出了代码中的值，部分有出入，但低位一致**)

```
WG_INT16:
        patch_0 = -9/16*64*32'h55550000 = -36'hB_FFF4_0000 = 'hF4_000C_0000 低位一致,高位全1 h8f4_000c_0000
        patch_1 = -(-3/16)*64*32'h55550000 + 6 = 36'h3_fffc_0006     低位一致,高位全0 h3d03_fffc_0006
                  (patch_1中因为-1的系数在计算时只做了取反，需要补上6个+1)
        patch_2 = -(-3/16)*64*32'h55550000 + 2 = 36'h3_FFFC_0002     低位一致,高位全0 h3d03_fffc_0002
                  (patch_2中因为-1的系数在计算时只做了取反，需要补上2个+1)
        patch_3 = -(1/16)*64*32'h55550000      = 36'hFE_AAAC_0000    低位一致,
```

```
    高位全1 hfe_aaac_0000
8                   (patch_3中因为-1的系数在计算时只做了取反，+2-(+2)+1-(+2)+1 =
    0次，恰好不用补)
9    WG_INT8 :
10          patch_0 = -9/16*64*16'h5500 = -20'hBF400 =  24'shF40C00 = 22'h340C00
       一致
11          patch_1 = -(-3/16)*64*16'h5500 +6 - 6*20'h8_0000 = 24'shD3FC06 =
    22'h13_fc06  一致
12                  (patch_1中因为-1的系数在计算时只做了取反，需要补上 6个+1 和 缺
    的6个符号位)
13          patch_2 = -(-3/16)*64*16'h5500 + 2 = 19'h3FC02    一致
14                  (patch_2中因为-1的系数在计算时只做了取反，需要补上2个+1)
15          patch_3 = -(1/16)*64*16'h5500 - 6*20'h8_0000 =
    24'hCE_AC00=22'h0E_AC00  一致
16                  (patch_3中因为-1的系数在计算时只做了取反，补 0个+1 和 6个符号
    位)
```

FP16

对FP16，res_tag位中每一个1代表一个5；64个8比特的res_tag经过简单加和后，根据WG矩阵计算求出对应的结果项。寄存数据时取反，变成减法，组装拼接将1还原为5，经过CSA压缩为WG对应的n0-n3，与对应的data的n0-n3一起求和，完成对结果的补全，即减去对应的mul中非标准booth的缺口。

上述INT8、INT16的分析中的第一条mul涉及的缺，在FP16中由res_tag完成；第二条WG下的-1导致的符号位缺口和+1，对FP16来说，相当于INT16，不存在符号缺，只缺相应个数的+1。

res_tag完成补全的过程中有哪些非标准做法引起的缺口呢？

考虑非WG：将64个b[0-7]加到一起为ps_n0b[0-7]_dc[6:0]，非wg下传给ps_n0b[0-7]，经过取反寄存，然后非wg下补0作为CSA树的输入。相应代码如下：

```
8559 assign ps_n0b0[6:0] = (cfg_is_wg_d1[8]) ? {1'b0, ps_n0b0_wg} : ps_n0b0_dc;
```

```
8910      ps_n0b0_d2 <= ~ps_n0b0;
```

```
9572 assign ps_n0_in_b0 = (~cfg_is_fp16_d2[0]) ? 8'b0 :
9573                      (cfg_is_wg_d2[4]) ? {~ps_n0b0_d2[6], ps_n0b0_d2} :
9574                      {1'b0, ps_n0b0_d2};
```

这里有两个地方存在非标准做法：

1. ps_n[0-3]b[0-7]在寄存时取反，目的是为了做减法，减去mul中引入的'h55550000('h5500对INT8)，但少了+1。在9624行组装时，为了将1还原成5，数据两次使用并被左移两位，因此少的1被放大为5。b0~b7的权重不同，b1比b0高4位，依次类推。9624行的拼接和9898行的拼接，移位都是做此对齐，因此**由取反未加1导致的缺口的总和是'h5555_5555**；

```
9623 // {2'b0,dat8,6'b0} {4'b0,dat8,4'b0} {6'b0,dat8,2'b0} {8'b0,dat8}
9624 assign ps_in_l1n0 = {2'b0, ps_n0_in_b1, 10'b0, ps_n0_in_b1, 10'b0, ps_n0_in_b0, 10'b0, ps_n0_in_b0};
```

```
9898 //chenhao: {{{dat14,24'b0}x2},{{6'b0,dat16,16'b0}x2},{{14'b0,dat16,8'b0}x2},{{22'b0,dat16}x2}
9899 assign ps_in_l2n0 = {ps_out_l1n3_1[13:0], 24'b0, ps_out_l1n3_0[13:0], 30'b0,
9900                      ps_out_l1n2_1, 22'b0, ps_out_l1n2_0, 30'b0,
9901                      ps_out_l1n1_1, 22'b0, ps_out_l1n1_0, 30'b0,
9902                      ps_out_l1n0_1, 22'b0, ps_out_l1n0_0};
```

2. 在进加法树前，加的是负数（已取反寄存），采用mul同样的思路，将符号位取反，使得进tree前全部为正数。如9572行所示，非wg下n0bx必然是正数，取反后符号位应是1，此处补的是0（wg下补的是符号位的反），因而 正确的数=ps_n0_in_b[0-7]-8'b1000_0000； 在后续的移位拼接过程中，bn导致的缺口就是(5*8'h80)<<4*n

```
9572 assign ps_n0_in_b0 = (~cfg_is_fp16_d2[0]) ? 8'b0 :
9573                      (cfg_is_wg_d2[4]) ? {~ps_n0b0_d2[6], ps_n0b0_d2}
9574                      {1'b0, ps_n0b0_d2};
```

由b0-b7导致的总的缺口是: -'b1010_1010_1010_1010_1010_1010_1010_10_1000_0000 = -'h2a_aaaa_aa80

FP16下的pathch_0，没有data的缺口，只考虑res_tag的两个缺口的和，如下，patch_1-3在非wg下为0。

```
1  FP16: patch_0 =  -'h2a_aaaa_aa80 + 'h5555_5555  = 'hD5_AAAA_AAD5 =
   38'h15_aaaa_aad5  一致（有效位）
```

考虑WG：res_tag依旧是两个地方缺数，且情况完全相同。取反时少1，n0-n3都有一样的拼接移位过程，缺数 'h5555_5555；进加法树前，wg下扩充的符号位是符号位的反，导致需要减去 8'h80 才是正确的数，n0-n3都涉及b0-b7，因此导致总的缺数依然是 -'h2a_aaaa_aa80；

```
1  WG_FP16: patch_0 =  patch_0_FP16 = 38'h15_AAAA_AAD5   一致（有效位）
```

patch_1/2/3与patch_0不同的地方在于，要考虑16位data计算时只取反导致的相应个数的+1的缺口。patch_1要额外+6，patch_2要额外+2，patch_3加的减的抵消了。符号位缺口，在FP16里不存在（只在INT8里出现）。

因此得到如下：

```
1   WG_FP16 :
2            patch_0 =  -'h2a_aaaa_aa80 + 'h5555_5555    = 38'h15_AAAA_AAD5   一
   致（有效位）
3            patch_1 =  -'h2a_aaaa_aa80 + 'h5555_5555 +6 = 38'h15_AAAA_AADB   一
   致（有效位）
4            patch_2 =  -'h2a_aaaa_aa80 + 'h5555_5555 +2 = 38'h15_AAAA_AAD7   一
   致（有效位）
5            patch_3 =  -'h2a_aaaa_aa80 + 'h5555_5555    = 38'h15_AAAA_AAD5   一
   致（有效位）
```

上述就是全部分析，如果考虑 非INT8 的有效位只有[37:0]，INT8 只有[21:0]和[45:24]，那么计算得到的patch和代码里一致！

但是以46位全长来考虑，计算在38位以上的高位和代码是有一定出入的！

请教了原作者，经确认这里只有低38位有效，所以高位无所谓，上述分析和计算是正确的。

BTW： mac代码里还有一个比较有意思的代码，如下，对INT8在扩位宽时，一个1'b0一个3'b0怎么解释？

```
9400      pp_in_l3n0_0 = ~cfg_is_int8_d2[0] ? {4'b0, pp_out_l2n0_0_d2} :
    {1'b0, pp_out_l2n0_0_d2[41:21], 3'b0, pp_out_l2n0_0_d2[20:0]};
```

原作者的解答：这里对于22位有效位，只要扩展一个符号位。低部分补充3个0是为了隔离。两个INT8运算共享相同的一个CSA Tree，低部分的int8进行加法的时候，不能让可能产生的无用进位污染高部分的INT8运算，所以需要补充若干个0进行隔离。这里通过计算确认进位不会超过3位，因此补了3个0.