量化模块报告

1. 硬件逻辑实现

首先先进行舍入操作。其逻辑如下：

对于一个正数，比如 010.1101，我们想要把小数部分变为 1 位，那我们要做的就是首先判断数为正还是为负，之后我们要判断是否需要进位，我们要看被截掉的部分中的最高的 bit，如果这个 bit 为 1，表明需要进位，比如上面的数进位后为 011.0。

对于负数，处理方法恰好相反，比如 100.1101，然后我们要看截断部分的最高位是否为 1，同时我们要看除最高位的其他位是否有 1（按位取或），若同时满足，对于负数来说是不用进位的，和正数不同，负数不进位是需要加 1 的。对于上面的数，因为是负数，同时被截取的三位（101）中最高位为 1 且还有其他的 bit 也包含 1，因此不需要进位，但不需要进位时需要加一，因此进位后结果为：101.0

```
assign carry = data_in[19] ? (data_in[7] & (
    data_in[6]|data_in[5]|data_in[4]|data_in[3]|
    data_in[2]|data_in[1]|data_in[0]) ): data_in
    [7];
```

以上代码实现了上面说的进位的机制，我使用了三元运算符来计算，首先判断正负，如果为正，那 data_in[7] 就是进位标志符，如果为负，在判断 data_in[7] 的同时还需要判断 0-6 位是否有 1。

在得到 carry 位后，我们需要将 carry 位加到原数据上，但是这里在进位时，会导致可能产生溢出，为了避免溢出的情况，我们需要对整数的部分进行一个符号位拓展，符号位为 data_in[19]，拓展后的结果还需要加上上面我们得到的 carry 位。由于 verilog 中位运算非常方便，所以这个操作可以写的很简洁。

```
assign Fix_13_1 = {data_in[19],data_in[19:8]} +
    carry ;
```

之后处理饱和的部分，对于正数的饱和，我们要把它饱和处理到最接近的最大正数，比如 01111.1 若要变成 Fix_4_1，那结果应该为 011.1。

对于负数的饱和，我们要把他饱和到最接近的最小负数，比如 10011.1 饱和后应该为 100.0。

从以上的部分可以发现，最接近的最小正/负数有统一的形式，为

$$sign\_bit, !sign\_bit, !sign\_bit...$$

因此实现逻辑就比较清晰了，如果越界的几位为全零（正数时）或全一（负数时），则表明没有越界不需要饱和，我们只需要截取低几位，即可作为我们的饱和操作后的值（可以看成逆向的符号位拓展）。如果越界了，那只需要将输出值设置为上面所说的数即可，这里也可以使用三元运算符使代码实现更为简洁。由于这里我是按每一位进行操作的，因此这里的代码比较冗余，在此不做展示了，但逻辑实现已经在这里说明了。

之后将得到的数据作为 data_out 加入数据流中即可。

下面的软件实现时，也采用了类似的逻辑。

2. 软件逻辑实现

由于输入数据分别为 Fix_8_5 与 Fix_8_4，因此第一步是用 int8 类型的数组读入输入特征图（66×66）与权重（3×3）。因为 python 中没有小数转为二进制的方法，因此我选择将 8bit 作为整体进行操作。在之后的量化时将 20bit 作为整体量化为 8bit。

验证特征图数据如下。

```
[[   0    0    0 ...    0    0    0]
 [   0  -71   44 ...  -36 -124    0]
 [   0    5    6 ...  -46  -50    0]
 ...
 [   0 -116 -120 ... -103    2    0]
 [   0   59 -104 ... -108   55    0]
 [   0    0    0 ...    0    0    0]]
```

验证卷积核数据如下。

```
[[ -74 -116  127]
 [  38   95  -46]
 [ -10  -23   52]]
```

因为乘累加结果会达到 20bit，因此需要对数组进行类型转换，转换为 int32。

定义卷积结果的矩阵，类型为 int32，大小为 64×64。接下来按照卷积的定义式，对输入特征图进行卷积操作。

接下来对所有的输出数据进行量化操作并写入文件中。量化逻辑如下：

首先判断正数或者负数，因为正负数的量化逻辑不同。接下来用按位与的操作取出第 8bit 的值，因为这个值决定了我们是否要进位。如果需要进位，那我们先把数据右移 8 位然后进位，先移位的原因是先进位有可能导致范围溢出。如果不需要进位，我们直接右移 8 位即可。之后对整数的部分进行饱和操作，如果 num>127，则饱和规约到 127。负数的处理和正数类似，但是在判断是否进位时我们还需要确定后七个 bit 均为 1，我们也是先处理小数再进行饱和。

```
def cutoff(num):
    if num >= 0:
        pn_flag = 0
    else:
        pn_flag = 1
    flag = num & 0x80
    if pn_flag == 0:
        if flag > 0:
            num = (num >> 8) + 1
        else:
            num = num >> 8
        if num > 127:
            num = 127
    else:
        if flag > 0 and num & 0x7F > 0:
            num = (num >> 8) + 1
        else:
            num = num >> 8
        if num < -128:
            num = -128
    return num
```

这里一定不能先处理饱和再处理小数，因为如果先处理饱和再小数进位，那很有可能让数据越界，此时想保证正确性一定要再进行一次饱和，这样就非常繁琐了。

这里的逻辑实现我一开始使用了先对两个输入转小数进行计算后截断再转回二进制数的方法，但是由于 python 中没有小数转二进制的内置函数，而且这样的方法逻辑非常复杂需要考虑的事情很多，因此最后使用了转成 8bit 的数进行操作的方法。

3. 结果比对

我编写了小的 python 代码，读入硬件和软件的输出，之后逐位检查，发现没有不同的 bit，表明结果正确。

```
flag = 0
for i in range (input_data.shape[0]):
    for j in range (input_data.shape[1]):
        if input_data[i][j] != input_d[i][j]:
            flag = 1
```

输出 flag，flag=0，表明两个文件中的值一致。

4. 思考题

4.1. 思考题 1

relu 的操作就是，对负数进行截断，将所有的负数全部截取为 0，所有的正数全部保留。因此 relu 的实现只需要对负数进行操作。代码如下：

```
if (data_out < 8'b0) begin
    data_out <= 8'b0
end
```

在 python 中的实现类似且更加简单，比起 Verilog，python 中可以实现更多的更复杂的激活函数如 leaky-relu，elu 函数，在实际应用中可能有更加好的效果。

4.2. 思考题 2

我查看了 cnn.v 文件的代码，在 Dconv_3×3_PE 模块中调用了 cutoff 中的 radix_point_right 参数，因此这个模块的实现依赖于这个参数，因此 cnn.v 模块并不支持小数点的位置变化。

cutoff.v 文件中的操作并不依赖于固定的小数点位置，因此我认为他可以支持小数点位置变化。

4.3. 思考题 4

python 的 round 函数的截断逻辑和我们实现的截断逻辑有所不同，我实验了一些数据，发现 python 对于一些数字，他们的被截取的地方的下一位是 5 并且这一位后再没有数字的时候，round 的处理是直接截取。比如 1.675 在 round 后会变成 1.67，这个即为两者不同之处。