1. MAF 硬件设计

假设 A、B、C 为三个单精度浮点数，MAF 要实现的操作就是：A*B+C。MAF 的分为四个计算阶段：乘法阶段、加法阶段、规格化阶段和舍入阶段。每个阶段主要实现的操作总结如下

- 乘法阶段：实现操作数 A 和操作数 B 24-bit 尾数（A_frac 与 B_frac）乘法，得到 2 个 48-bit 的 carry 和 sum。同时根据三个操作数指数部分的关系对操作数 C 的尾数(C_frac)部分进行对齐移位操作，便于进行后续的加法操作。
- 加法阶段：将 A_frac 与 B_frac 相乘得到的 2 个结果与对齐移位后的 C_frac 进行相加得到尾数部分的精确结果；同时执行前导零检测逻辑得到用于后续规格化移位的数量。
- 规格化阶段：根据前导零检测的结果对尾数部分进行移位并得到尾数部分的 round bit、guard bit 和 sticky bit,得到 27 位规格化的尾数部分用于下一阶段的舍入操作。
- 舍入阶段：根据规格化的输出和舍入模式的选择，对 27-bit 尾数进行舍入得到最终结果的 24-bit 尾数部分。

根据 PowerISA2.06B 的定义，Vector 浮点类指令需要支持两种模式：java mode 和 non java mode。两种模式的主要不同在于对待非规约化数的情况上，对于 non java mode 非规约化数值当成零来处理。java mode 下数值的计算精度会更高。

1.1乘法阶段具体实现

在乘法阶段主要有两大操作：A_frac 与 B_frac 的乘法和 C_frac 的对齐移位操作。在这里 A_frac 与 B_frac 的乘法算法选择传统的乘法算法，即先产生 24 个部分积，然后对部分积进行相加得到 2 个 48-bit 的 carry 和 sum 部分。在尾数 C_frac 的对齐移位操作，则采用只对 C_frac 进行右移对齐。乘法阶段结束后会得到 2 个 48-bit 的 carry 和 sum 部分，同时 C_frac 对齐移位后会得到一个 98-bit 的移位后结果（high 26bits + middle 48 bits+low 24bits）。指数部分和尾数部分也会进行相应处理。

1.1.1 unpackage 操作

在进行乘法阶段的操作之前要先结合是否为 java mode 对输入的操作数进行解析，也即是 unpackage 操作。用 1-bit nj_mode 信号来表明当前操作处在何种模式之下（1—non java mode, 0—java mode）。

如图 1-1 所示为 unpackage 主要的操作在于 24-bit 的尾数选择上。如果输入操作数的指数部分为 8’h0 那么该操作数为一个非规约化的数值，图 1-1 所示用 denorm 信号来标识操作数是否为一个非规约化数值（1 表示该操作数是规约化数值，0 表示该操作数为规约化数值）。如果该操作数为一个非规约化数值需要根据 nj_mode 信号来选择尾数值，以操作数 A 为例（B、C 类似），尾数部分值选

择规则如下：

{denorm_A, nj_mode}	尾数部分值	说明
2’b00	{1’b1,A[22:0]}	规约化数值，不管是否为 java mode，其隐含的前导整数部分都为 1.
2’b01	{1’b1,A[22:0]}	同上
2’b10	{1’b0,A[22:0]}	java mode 下，非规约化数值隐含的前导整数部分为 0.
2’b11	24’h0	non java mode 下，非规约化数值当成零来处理

对于指数部分，非归约化数值的指数值为-126，就不能按照归约化数值的指数部分的计算方法（指数部分-偏移量）进行计算了，指数部分值的选择规则如下：

{denorm_A, nj_mode}	指数部分	说明
2’b00	A[30:23]	规约化数值，不管是否为 java mode，其指数部分都为 A[30:23].
2’b01	A[30:23]	同上
2’b10	8’h1	java mode 下，非规约化数值的指数部分为 -126.
2’b11	A[30:23]	non java mode 下，非规约化数值当成零来处理

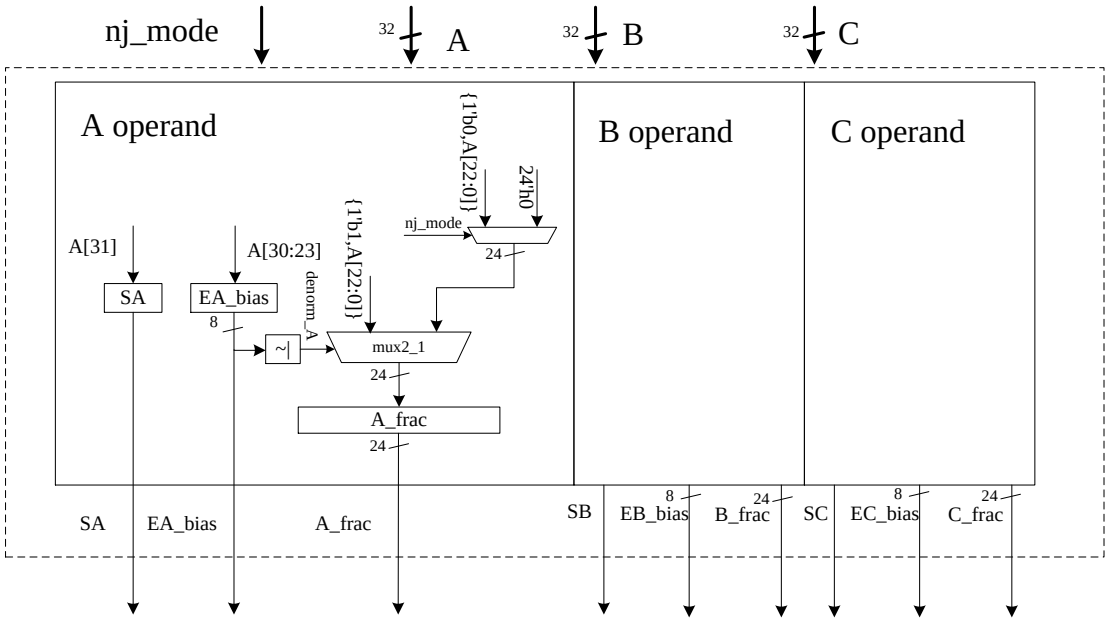

图 1-1 对输入操作数的 unpackage 操作

1.1.2 A_frac 与 B_frac 的乘法

24-bit 的 A_frac 与 B_frac 的乘法算法采用传统的乘法算法，即先产生 24 个部分积，然后部分积相加得到 carry 和 sum 部分，与 Booth 乘法不同，产生的 carry 和 sum 是两个无符号数。由于算法简单，因此不在此进行算法的详细描述。

1.1.3 C_frac 的对齐右移

在进行操作 A*B+C 时需要考虑到，A、B、C 三个都是单精度浮点数，其符号位可正可负，但总的来讲可归结为两种情况

- 一种是 A*B 的结果的符号与 C 的符号相同，可能的情况是 A,B,C 全部为正数或者 A,B,C 全部为负数。
- 另一种是 A*B 的结果符号与 C 的符号不同，可能的情况是-|A*B|+|C|或者是|A*B|-|C|。

在实际进行硬件设计的时候只需要考虑|A*B|+|C|和|A*B|-|C|这两种情况即可，至于符号部分可以另作处理。如果是后一种情况就会涉及对 C_frac_inv 求补的操作。求补操作涉及按位取反和末位加一操作，通常末位加一的操作会放到加法阶段进行，作为补偿进位位处理。

为了减少计算延迟，尾数 C 的按位取反和对齐移位操作与尾数 A 和尾数 B 的乘法操作是并行进行的。通过将 C_frac_inv 放到 A_frac 与 B_frac 的乘法结果的最高位的左边，在进行对齐移位的时候只对 C_frac_inv 进行右移即可，如图 1-2 所示。在 A_frac * B_frac 与 C_frac_inv 之间插入两个额外的比特位是当 C_frac_inv 不移位的时候能准确进行舍入。对齐右移的依赖于值 $d=E_C-(E_A+E_B)$，E_A、E_B、E_C 分别为源操作数 A、B、C 的指数。

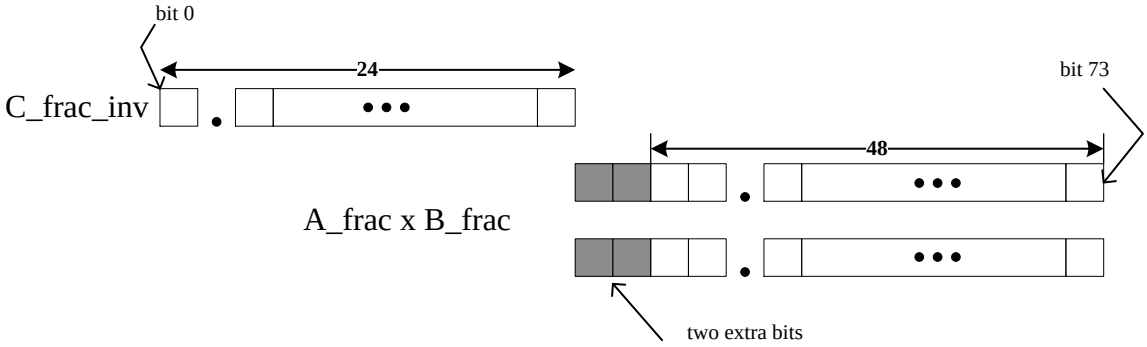

图 1-2　　C_frac_inv 的对齐移位

分两种情况讨论：

1） d >= 0.在传统的对齐移位方案中，A_frac * B_frac 会右移 d 位。最多移位 27 位，因此当 d>=27 时，A_frac * B_frac 会放在 C_frac_inv 最低有效位的右侧；在这种情况下，A_frac * B_frac 仅仅影响 sticky bit 的计算。与之相对应，采用这样实现方式需将 C_frac_inv 右移 27-d 位，并且移位的数目=max{0,27-d}。

2） d < 0.在传统的对齐移位方案中 C_frac_inv 应该右移 d 位。在这种情况下，最

多右移 47 位，对于 d <= -47 的情况，操作数 C_frac_inv 会被放置在 A_frac * B_frac 结果最低有效位的右侧，仅仅影响 sticky bit 的计算。与之相对应，C_frac_inv 应该右移 27-d 位，那么此时的移位数目=min{74,27-d}。

综合两种情况，移位的数目在[0,74]之间，因此需要一个 74-bit 的右移位单元。而且移位的数目=27-d。移出范围之外的数（低 24bit）主要用来计算部分 sticky bit，该部分得到的 sticky bit 会用来计算最终的 sticky bit。

另外如果是一个等效的乘加操作，则在移位之前需要将 C_frac_inv 的低 74bit 全部填充零，移位后的高位部分也填充零。如果是一个等效的乘减操作，那么在移位之前需要将 C_frac_inv 的低 74bit 全部填充 1，移位后的高位部分填充 1.对于 d<0 的情况，对于等效的乘减操作来讲，如果低 24bit 全部为 1，那么得到的中间 st1（粘滞位）就应该为 0，且后续进行 3:2CSA 操作的时候需要在有一个补偿进位位.如果不全为 1 那么 st1 就应该为 1，且后续的 3:2CSA 操作中就不需要补偿进位位。

对齐移位后会得到一个 98-bit 的移位后结果，该结果低 24 位用来计算 st1，中间 48 位会与乘法得到的 carry 和 sum 进行相加。而高 26 位则保持不变。如图 1-3 所示为乘法阶段的硬件结构图。

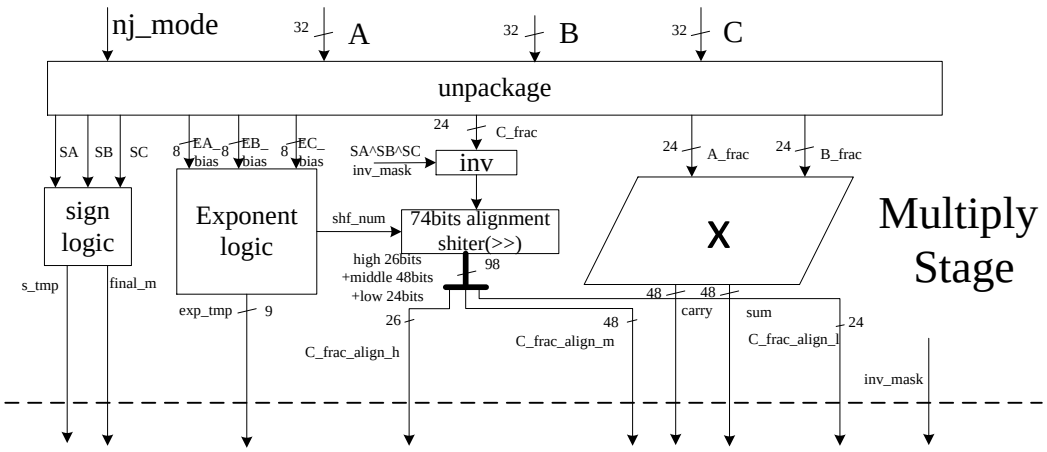

图 1-3　　乘法阶段的硬件结构图

unpackage 硬件结构图已经在 1.1.1 中做了详细说明，而传统乘法的的硬件比较简单，因此就不单独介绍了。图 1-3 中采用信号 inv_mask 来标示所要进行的操作是一个等效的乘加操作还是一个等效的乘减操作（1—乘减操作，0—乘加操作），而 inv_mask=SA^SB^SC。

接下来需要介绍一下如何产生右移移位个数的指数部分逻辑和 74-bit 的右移移位模块。

1.1.4　指数部分移位逻辑及特殊情况处理

由 1.1.3 知，$d=E_C-(E_A+E_B)$，右移移位个数用 7-bit 信号 shf_num 标示，通过总结可以得到

- 当 d>=27 时，shf_num = 0

- 当 d<=-47 时，shf_num= 74
- 当-47<d<27 时，shf_num=27-d

在得到 d 之后需要进行两次减法计算，26-d 和 d-(-46)，即 d+46。根据结果符号分别判断 d 与 27、-47 之间的大小关系。前者符号位为 1 表示 d>=27，后者符号位为 1 表示 d<=-47.如图 1-4 所示为产生 shf_num 的硬件结构图。EA，EB，EC 分别为去偏移量（偏移量为 127）后的指数部分。s_sel0 和 s_sel1 分别为计算 27-d 和 d+47 得到的符号位。shf_num 从 27-d、0、74 三个数中根据 s_sel0 和 s_sel1 的值进行选择，shf_num 为一个 7-bit 的值（最大值为 74）。选择规则如下

{s_sel0,s_sel1}	shf_num
2'b00	26-d
2'b01	74
2'b10	0
2'b11	不可能出现的情况

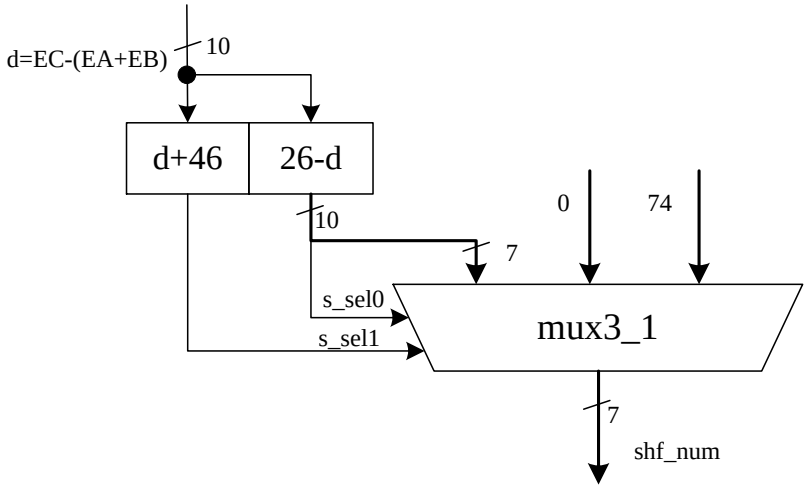

图 1-4　　产生 shf_num 硬件结构图

1.1.4.1 d 的取值范围

d=EC-(EA+EB)，其中 EA,EB,EC 的取值范围为[-127,127]（其中指数为 128 时的情况需要单独进行处理），理论上来讲，EA+EB 的取值范围为[-254,254]，那么理论上 d 的取值范围就为[-381,381]。在计算出 EA+EB 的取值之后需要判断两种特殊情况：

1. 通过与-126 进行比较，判断乘法是否存在下溢，如果 EA+EB 小于-126 则发生下溢，并将下溢标志位 underflow_m（ask）置位。
2. 通过与 127 进行比较，判断乘法是否存在上溢，如果 EA+EB 大于 127 则发

生上溢，并将上溢标志位 overflow_m(ask)置位。

1.1.4.2 指数部分运算过程中特殊情况的考虑

表 1-1　浮点数分类

形式	指数	小数部分
零	0	0
非规约形式	0	非 0
规约形式	1 到 2^e-2	任意
无穷	2^e-1	0
NaN	2^e-1	非零

表 1-1 所示为浮点数分类，其中 NaN 又分为两类：Signaling NaN 和 Quiet NaN。前者 23-bit 尾数的 MSB 为 0，后者为 1。

通常情况下，QNaN 不会产生异常，而 SNaN 会产生异常.

在该小节中将特殊情况分为两类，一类是能够产生异常的特殊情况，也可称为异常情况，另一类是不产生异常的特殊情况。先看第一种特殊情况。

第一种特殊情况

在《PowerISA_V2.06B》，Book I，Chapter6 中关于 Vector 浮点部分定义了六种异常情况及其相关处理方式。六种异常情况如下

- NaN Operand Exception
- Invalid Operation Exception
- Zero Divide Exception
- Log of Zero Exception
- Overflow Exception
- Underflow Exception

详细的异常说明及处理见指令集说明(《PowerISA_v2.06B》p.201)。

对于 MAF 操作来讲，可能出现的异常情况及其处理总结如下

1. 任一输入操作数（VRA 或 VRB 或 VRC）为 NaN 时，其结果为输入的那个 NaN。
2. 两个无穷的绝对值做差（∞-∞），其结果为 QNaN，0x7fc0_0000。
3. 无穷与零相乘（0*∞），其结果为 QNaN，0x7fc0_0000。
4. 当出现上溢的时候，其结果为无穷，符号与中间结果的符号相同。
5. 当出现下溢的时候，如果是 non java mode，则结果为零，符号与中间结果的符号相同。如果是 java mode，那么中间结果经过舍入之后会得到一个非规约化数值。

对于第 1 种情况的判断及处理都相对容易。

第 2 中情况，只有当当前的操作是一个等效的乘减操作，且 A 与 B 之间有一个为无穷，另一个不为零，操作数 C 也为无穷的时候才会出现该种情况。

对于第 3 种情况的判断也相对容易。
对于第 4 种情况和第 5 种情况需要在计算的过程中才能发现。

对于第一种特殊情况的处理分为两种方式，一种是某些特殊情况需要特殊处理，即在计算的主流水线之前另设特殊情况的单独处理流水线，如果检测到特殊情况，那么最后的结果就用特殊流水线情况下得到的值。另一种就是在计算的主流水线对特殊情况进行检测和处理。

其中，第 1,2,3,4 种特殊情况需要单独的模块进行处理，对于第 5 种情况，如果是在 non java mode 下，如要单独处理，而如果是 java mode 下则需要纳入主流水线中进行检测和处理。

第二种特殊情况

再看第二种特殊情况，这里把第二种特殊情况定义为，除了第一种特殊情况外，只要是输入操作数中有不是 IEEE754 中定义的规约化数值的情况都可称之为第二种特殊情况。主要包括以下几种情况：

- 输入操作数中有零。
- 输入操作数中有非规约化数。
- 输入操作数中有无穷数值。

对于第二种特殊情况的处理也分为两种，一种是特殊情况特殊检测、特殊处理，另一种是纳入主流水线进行检测和处理。接下来需要分清哪些特殊情况需要特殊处理，而哪些特殊情况需要纳入主流水线中进行处理。

对于操作数中有零的情况，

1) 如果 A 或 B 中其中任意一个为零，则结果就为 C，该特殊情况需特殊处理
2) 如果 A、B 都不为零，而 C 为零，则需要纳入主流水线中进行处理，在主流水线中需要提前对操作数 C 进行判断，并进行相应处理（应该不用刻意进行判断）。

对于操作数中有非规约化数的情况，在 java mode 下纳入主流水线进行处理，在 non java mode 下当成零来处理。

对于操作数中有无穷的情况，这种情况下都需要单独进行处理。该种情况可能出现的特殊情况及其处理总结如下

1) 3 个操作数中只有一个为无穷，如果 A 为无穷且 B 不为零，则结果为无穷，符号与 A,B 操作数的符号相关；B 为无穷也同样进行处理；如果 C 为无穷，结果为 C。
2) 3 个操作数中有 2 个无穷，如果 A 不是无穷且不为 0，而 B、C 为无穷，那么 A*B 就为无穷，如果 A*B 得到的结果符号与 C 符号相同，那么结果为无穷，符号就是两者的符号；如果 A*B 得到的结果符号与 C 符号不同，那么就出现了异常，其结果为 QNaN。同理如果 B 不是无穷且不为零，也进行此种处理。如果 C 不是无穷，而 A,B 为无穷，那结果就为无穷，结果的符号与 A,B 的符号相关。
3) 3 个操作数都是无穷，那么结果为无穷，结果符号与 A,B 的符号相关，而与 C 的符号无关。

1.1.5 74-bit 对齐移位单元

对 C_frac_inv 的右移移位操作需要一个 74-bit 的移位器，在进行移位之前首先要对 24-bit 数 C_frac_inv 进行补齐操作，如果当前的操作是一个等效的乘法操作则在 C_frac_inv 补齐 74-bit 0，如果当前的操作是一个等效的乘减操作，则在对 C_frac_inv 补齐 74-bit 的 1.

根据 shf_num 对补齐后的的 C_frac_inv 进行右移，如果当前的操作是一个等效的乘加操作则高位部分填充 0，如果当前的操作是一个等效的乘减操作则高位部分补充 1.移位之后就得到了 98-bit 的结果 C_frac_align.

得到的移位后的 98-bit 的结果 C_frac_align 分成三部分：高 26-bitC_frac_align_h，中间 48-bitC_frac_align_m 和低 24-bitC_frac_align_l。其中 24-bit 信号 C_frac_align_l 主要用来计算中间结果的 sticky bit，不妨记为 st1.如果当前操作为一个等效的乘加操作，那么将 24-bit 的 C_frac_align_l 按位或就可得到 st1；如果当前操作是一个等效的乘减操作，那么如果 C_frac_align_l 全为 1，则 st1 为 0，并且在下一个加法阶段需要一个有效的补偿进位位（对 C_frac 求补后的补偿位），如果 C_frac_align_l 不全为 1，则 st1 为 1，并且在下一个加法阶段不需要有一个有效的进位补偿位。中间 48-bit 的 C_frac_align_m 主要用来跟乘法得到的 carry 和 sum 部分进行累加（先经过一个 3:2CSA，再经过一个 CLA）。得到 48-bit 的尾数低位部分加法结果记为 frac_inter_m，而尾数高位部分的加法结果依赖于 26-bit 的 C_frac_align_h。需要对高 26-bit 的 C_frac_align_h 增加一个符号位，如果当前的操作为一个等效的乘加操作那么该符号位为 0，否则为 1.不妨记增加符号位的 C_frac_align_h 为 C_frac_align_h_signed，C_frac_align_h_signed 为 27-bit，如果低位部分无进位，那么尾数结果的高位部分就是 C_frac_align_h_signed，如果低位部分有进位，那么尾数结果的高位部分就是(C_frac_align_h_signed+1)，记高位部分的结果为 frac_inter_h,那么计算之后尾数部分的中间值就是 frac_inter_tmp={frac_inter_h,frac_inter_m,st1}，该值是一个有符号数，如果是负数（最高位为 1）需要进行求补得到真值 frac_inter。

1.1.6 符号和指数部分的处理逻辑

符号部分

通过信号 inv_mask 我们可以得知当前的操作是一个等效的乘加操作还是一个等效的乘减操作，但是不能确定最终结果的符号是正还是负，因此需要符号检测逻辑来判断最终结果的符号。符号检测逻辑的判断规则如下：

- C 符号为正，且 A,B 同号，那么最终结果的符号为正。
- C 为正，且 A,B 异号，此时需要结合加法阶段尾数的结果符号位进行判断，如果尾数的中间结果符号为正，那么最终结果符号为负，否则为正。
- C 为负，且 A,B 异号，那么最终结果符号为负。
- C 为负，且 A,B 同号，此时需要结果加法阶段尾数的结果符号位进行判断，如果尾数的中间结果符号为正，那么最终结果符号为正，否则为负。

在乘法阶段会得到一个中间符号 s_tmp，同时得到一个标志位 final_m，如果此标志位为 1 表示 s_tmp 就是最终的符号位，如果为 0 表示 s_tmp 不是最终的符

号位，需要结合尾数的中间结果的符号得到最终结果的符号。

指数部分

乘法阶段得到的指数部分有两个可能：一个是 A*B 之后的指数部分，另一个是 C 的指数部分。通常情况下会取两者中较大者作为乘法阶段的的指数部分。需要注意的是在乘法阶段我们提前对操作数 C 的尾数部分进行左移了 27-bit，相当于对乘法得到的 carry 和 sum 右移了 27-bit，在乘法阶段，经过 74-bit 的移位单元后得到了 98-bit 的移位后的结果，假设最高位为整数位，取 exp_tmp 的值为 **EC** 与 **EA+EB+27** 两者中的较大者。该指数部分数值在规格化和舍入阶段会进行调整从而得到最终结果的指数部分。

1.2加法阶段的具体实现

如图 1-5 所示为加法阶段的硬件结构图，符号部分和指数部分在该阶段未做任何处理，因此并没有在硬件结构图上画出。具体操作流程已在 1.1.5 节进行了描述，这里就不再赘述。

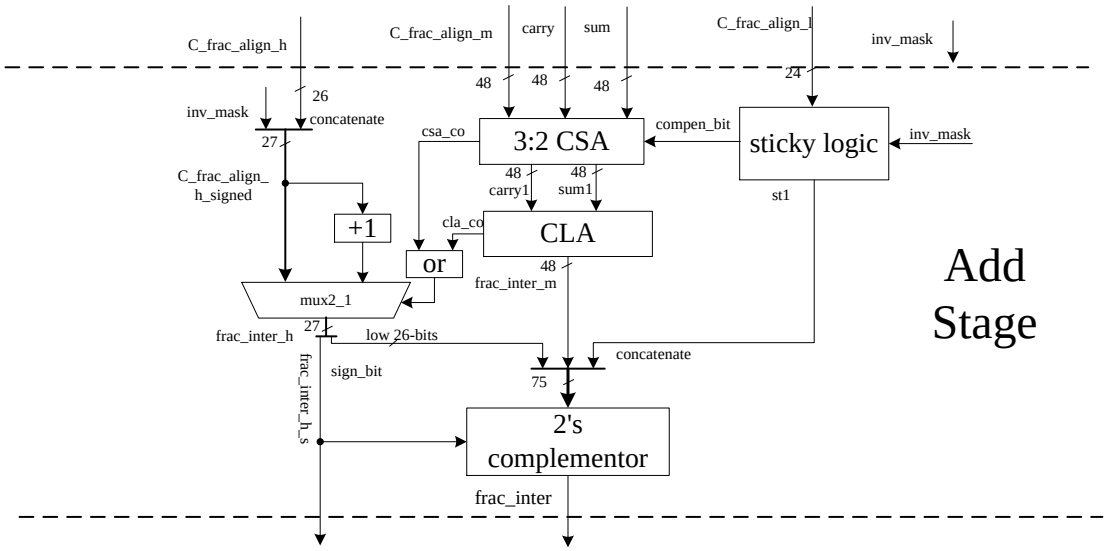

图 1-5　　加法阶段的硬件结构图

1.3规格化阶段的具体实现

规格化阶段的操作总结如下

- 尾数部分：先对加法阶段得到的尾数的中间值 frac_inter 进行前导零检测，并根据检测结果对中间结果尾数进行规格化移位，使得最高位整数部分为 1，根据移位的结果还需要对指数部分进行调整。之后求得中间结果的 guard bit,round bit 和 sticky bit 从而得到 27-bit 的尾数值，用于后续的舍入阶段进行舍入。
- 指数部分：在前两个阶段得到了指数部分中较大的那个的真值（A*B 和 C 的指数部分）exp_tmp，需要注意的是在乘法阶段我们提前对操作数 C 的尾数

部分进行左移了 27-bit，相当于对乘法得到的 carry 和 sum 右移了 27-bit，在乘法阶段，经过 74-bit 的移位单元后得到了 98-bit 的移位后的结果，假设最高位为整数位，取 exp_tmp 的值为 **EC** 与 **EA+EB+27** 两者中的较大者，其去值范围为[-127,154]。假设前导零检测的结果为 lz_count（7-bit 正数），调整后的指数为 exp_norm=exp_tmp-lz_count，其取值范围为 [-201,127]（lz_count==75 时特殊考虑，如下）

- 符号部分：结合 frac_inter_h_s 信号，并根据前 2 个阶段得来的 s_tmp 和 final_m 信号得到最终结果的符号为 s_final。具体检测逻辑见 1.1.6 所示。

如图 1-6 所示为规格化阶段的硬件结构图，在该阶段还需要判断如下特殊情况

1. 75-bit 的尾数全为 0，此时最终结果为 0，符号由符号逻辑部分确定。如果出现这种情况，将标志位 zero_m 置 1.
2. 规格化后的指数部分真值<=-127，则将标志位 denorm_m 置位。并计算非归约化尾数右移移位的个数 denorm_shf_num=|exp_norm+127|，如果 denorm_shf_num > 27，则移位个数为 27.

如图 1-6 所示为规格化阶段的硬件结构图，图中 denorm logic 见图 1-7 所示

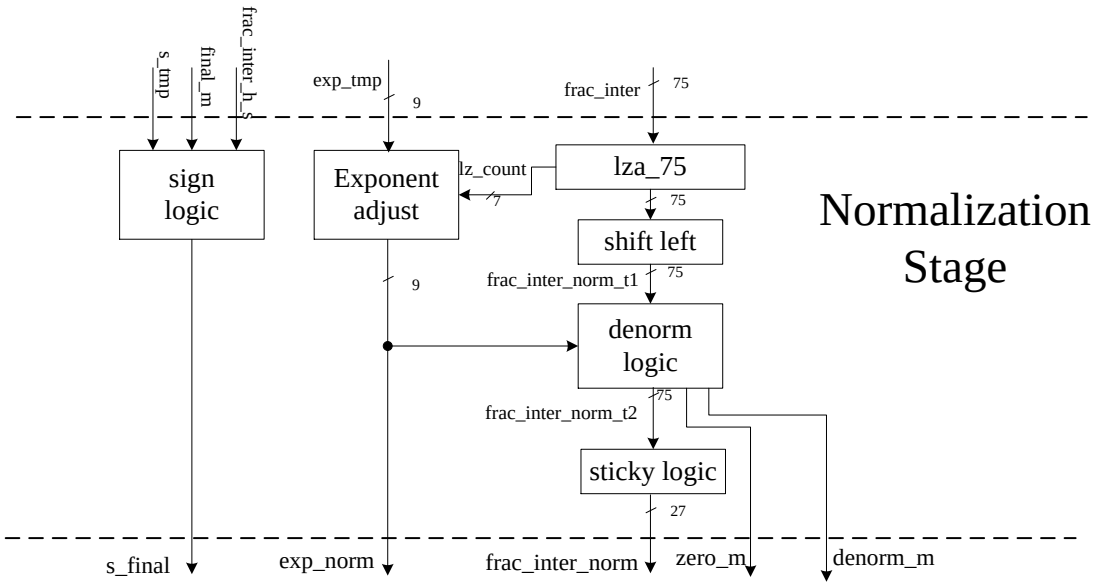

图 1-6　　规格化阶段的硬件结构图

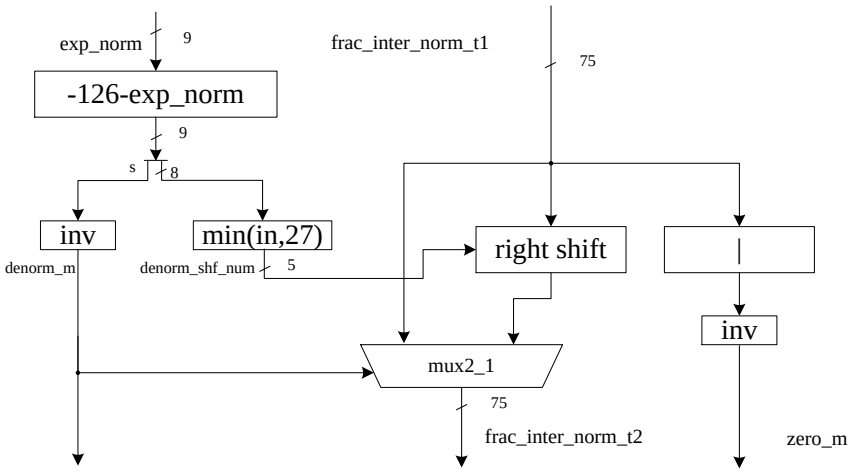

图 1-7　　denorm logic 硬件结构图

1.4舍入阶段的具体实现

在该阶段进行舍入，舍入模式为就近舍入，如果出现 tie 的情况，则舍入到偶数结果。经过规格化阶段之后，得到了 27-bit 的尾数中间值 frac_inter_norm（就是图 1-8 中的 Z），那么 Z2=frac_inter_norm[26:3]，Z1=frac_inter_norm[26:3]+1。就近舍入就是从 Z1 和 Z2 当中选择一个跟 Z 的绝对值之差更小的数作为最终结果，如果两者跟 Z 的绝对值差相等，那么选择 LSB 为 0 的作为最终尾数的结果。在该过程中可能还涉及对尾数部分的调整（Z1 有可能出现溢出的情况）。

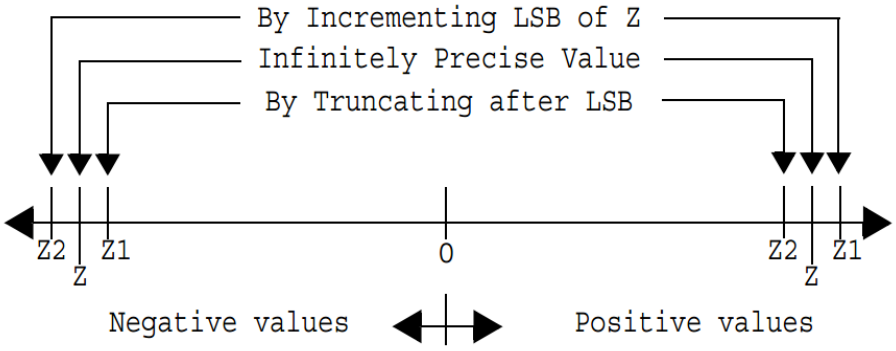

图 1-7　　Z1 和 Z2 的选择

如图 1-8 所示为舍入阶段的硬件结构图，指数部分的逻辑如下所示。

{denorm_m,overflow_round}	指数结果
2’b00	exp_norm+127
2’b01	exp_norm+128
2’b10	0
2’b11	1

舍入阶段之后再将各个部分进行打包，就得到了最终 32-bit 单精度浮点数结果了。

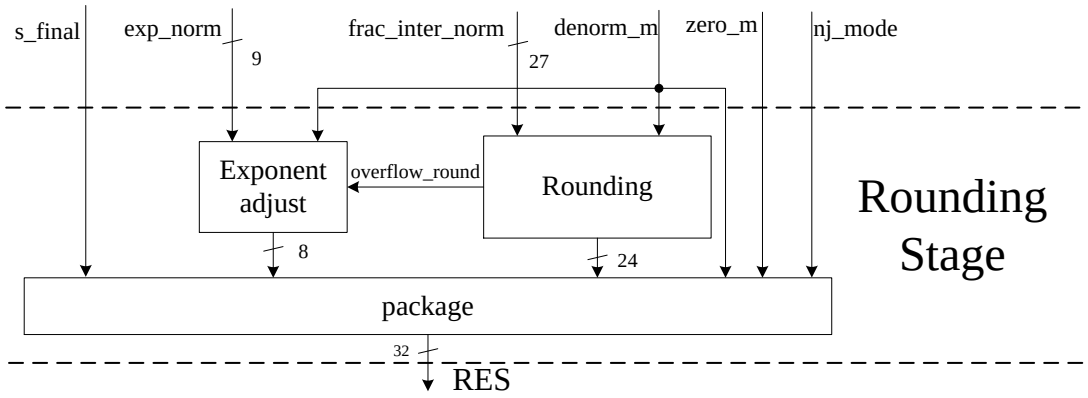

图 1-8　　舍入阶段的硬件结构图

1.5 主流水线的硬件结构框图

在 1.1,1.2,1.3,1.4 中分别给出了每个阶段的硬件结构框图，在该小节中会给出主流水线的硬件结构框图，在下小节中则会给出整个 MAF 的硬件结构框图。如图 1-9 所示就是主流水线的硬件结构框图。

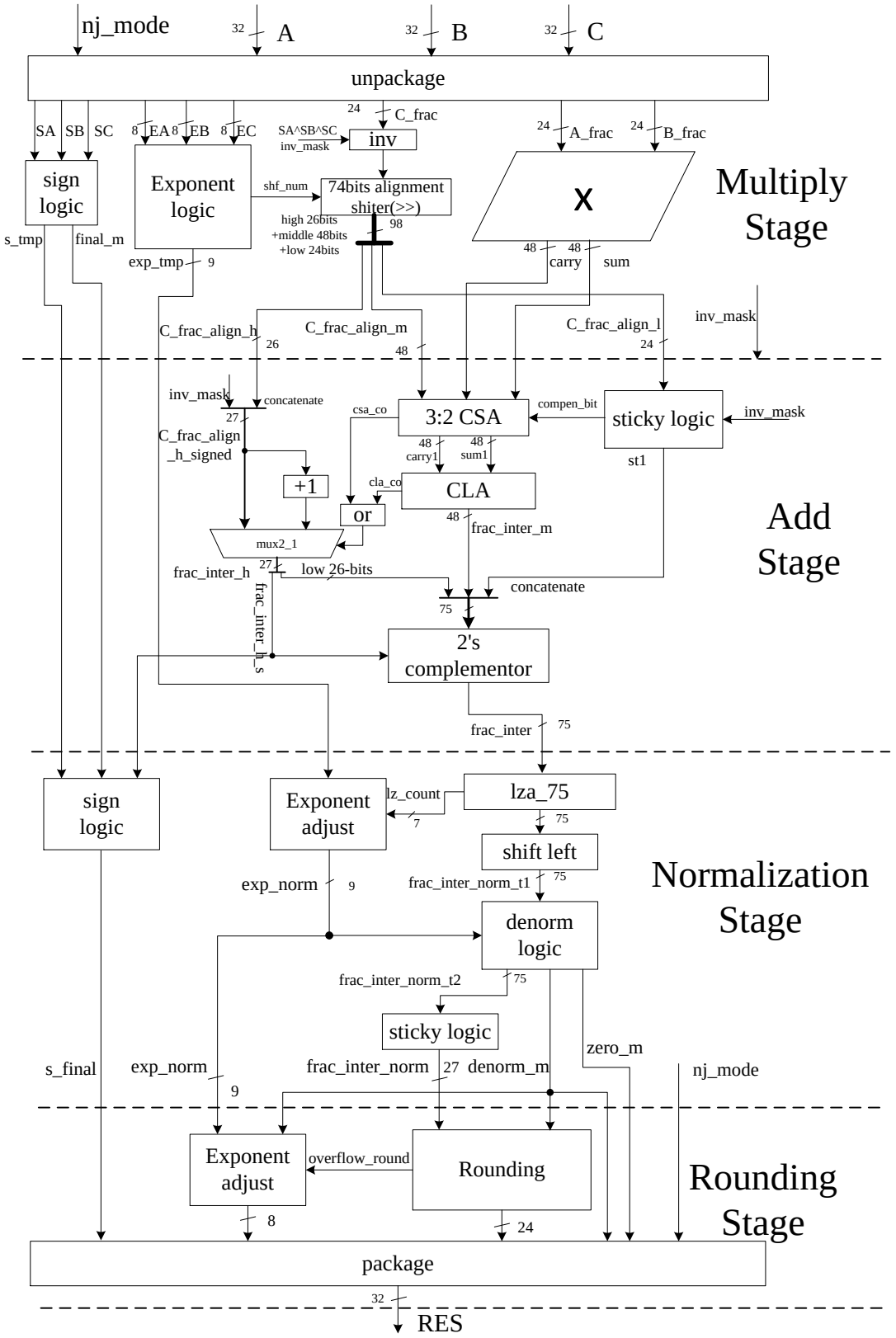

图 1-9　　MAF 主流水线硬件结构图

1.6MAF 整体硬件结构图

如图 1-20 所示为 MAF 的整体硬件结构图，最终的结果从主流水线产生的结果和特殊情况处理产生的结果中选择其中一个作为最终的结果，在特殊情况处理流水线中会检测输入的操作数是否为特殊情况，如果是，则为置位标志位 spec_m。最后根据 spec_m 从两个流水线中选择一个作为最终结果。对于特殊情况的检测及处理已经在 1.1.4.2 中进行了详细的说明，这里不再赘述。

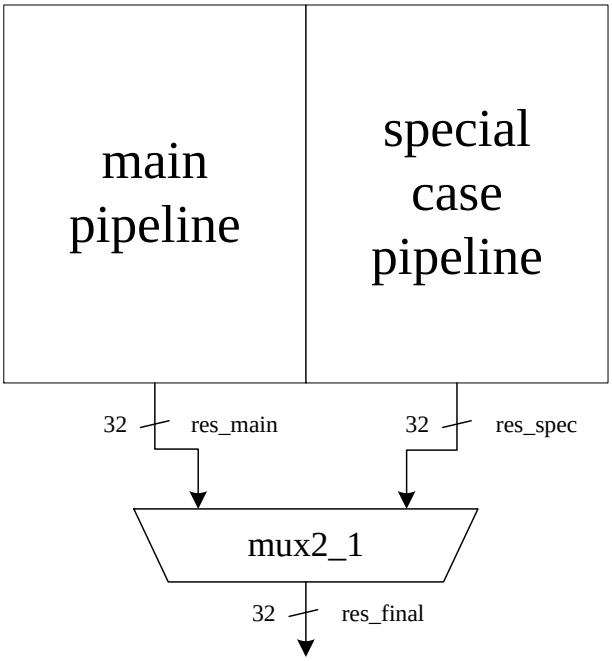

图 1-20　　MAF 整体硬件结构图