

part2 HiFi2 Feature

[返回目录位置](#)

HiFi 2/EP DSP包含一组8个48位操作数寄存器，AE_PR，如下图2-1所示。每个寄存器可以容纳一个或两个 24 位或 16 位操作数，具体取决于软件如何使用寄存器。**寄存器的两半始终是单独的数据项。例如，如果左移，L 元素不会溢出到高元素中。**

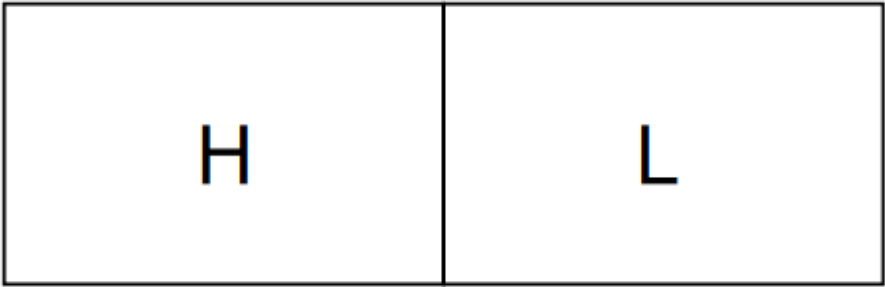


Figure 2-1 AE_PR Register

H和L长度都是24位，然后H和L里面都可以放置24位或者16位的数据。

当寄存器存储到内存时，寄存器的上半部分始终存储在较低的存储器地址中，而不考虑字节序。这使得相同的源代码可以在大端和小端处理器上运行。访问AE_PR寄存器的各个元素的操作引用助记符中带有选择器 L 和 H 的元素。

上面 “当寄存器存储到内存时，寄存器的上半部分始终存储在较低的存储器地址中，而不考虑字节序” 这句话我有点不理解？寄存器存储的位置是可以变化的吗？

这个可不可以理解成寄存器使用store类型指令时，H部分放置在内存的低位，然后L部分放在在内存的高位？比如内存的地址是0-48，那么H部分放置在0-23，然后L部分放置在24-47。

HiFi 2/EP还包含四个56位累加器寄存器AE_QR，通常保存乘法累加。每个寄存器还可以保存 32x16 位和 EP 上使用的 32x24 位乘法指令使用的 32 位操作数。**无论哪种方式，寄存器都保存单个数据项。**

HiFi 2/EP包含表2-1和表2-2中描述的一组专用状态寄存器。

Table 2-1 DSP Subsystem State Registers

State Register	Bit Size	Description
AE_OVERFLOW	1	指示自上次将算术运算重置为零以来是否有AE_OVERFLOW已饱和。
AE_SAR	6	包含各种 DSP 左移和右移操作的移位量。
AE_CBEGIN0 (EP only)	32	包含循环缓冲区的起始地址。
AE_CEND0 (EP only)	32	包含循环缓冲区的结束地址。

下面的状态寄存器属于DSP的位流和可变长度encodedecode支持子系统。一般来说，程序员不需要关心指令如何使用这些状态寄存器的细节，但是为了完整起见，这里记录了状态寄存器。对于已经比较熟悉可变长度encodedecode指令的读者来说，这些描述更有意义。

Table 2-2 Bitstream and Variable-length Encode/Decode Support Subsystem State Registers

State Register	Bit Size	Description
AE_BITH EAD	32	包含比特流头部的位。高半部分具有当前的 16 位，低半部分具有接下来的 16 位。只有高半部分用于输出比特流。
AE_BITP TR	4	在比特流头的 16 个最高有效位内偏移。对于输入比特流，此值表示应用程序已使用的最有效AE_BITHEAD位数。对于输出比特流，此值表示已初始化的最有效AE_BITHEAD位数。
AE_BITS USED	4	包含可变长度编码/解码指令在最后一个表查找中消耗或产生的位数。此值以二进制编码，但所有零被解释为值 16。
AE_TABL ESIZE	4	包含比可变长度解码的当前解码表大小的 2 底数小 1。0 对应于 2 个条目表;15 对应于 65536 条目表。
AE_FIRS T_TS	4	包含查找表层次结构中第一级的正确值 AE_TABLESIZE。此状态是一种优化，因此使用同一码本的连续解码操作之间不需要AE_VLDSHT指令。
AE_NEX TOFFSET	27	此状态用于三种不同的事情： 1. 在可变长度解码中：在AE_VLDL16T或AE_VLDL32T指令之前，AE_NEXTOFFSET是当前要查找的比特流前缀对应的表条目的索引。 2. 在AE_VLDL16T或AE_VLDL32T指令之后，AE_NEXTOFFSET是下一个解码查找表的基数的偏移量。 3. 在可变长度编码中：在AE_VLEL16T或AE_VLEL32T指令之后，AE_NEXTOFFSET的低位保存最近查找产生的码字位。
AE_SEA RCHDO NE	1	此状态告知AE_VLDL16C指令准备AE_NEXTOFFSET（使用 AE_FIRST_TS）以便从解码层次结构中的第一个表开始进行新的解码搜索。此状态是一种优化，因此使用同一码本的连续解码操作之间不需要AE_VLDSHT指令。

专用状态寄存器按如下方式分组为四个用户寄存器，以便进行高效的保存和还原操作：

```
1 user_register AE_OVF_SAR 240 { AE_OVERFLOW[0], AE_SAR[5:0] }
2 user_register AE_BITHEAD 241 AE_BITHEAD[31:0]
3 user_register AE_TS_FTS_BU_BP 242 { AE_TABLESIZE[3:0], AE_FIRST_TS[3:0],
  AE_BITSUSED[3:0], AE_BITPTR[3:0] }
4 user_register AE_SD_NO 243 { AE_SEARCHDONE[0], AE_NEXTOFFSET[26:0] }
```

对于每个user_register，都提供了相应的 RUR-、WUR- 和 XUR- 指令，用于读取、写入或交换一条指令中列出的所有状态。除了 RUR、XUR 和 WUR 系列指令外，还提供了具有相同签名的指令来读取和写入各个状态寄存器，如表 2-3 所示。

Table 2-3 State Register Access Instructions

Instruction	Intrinsic	Description
RUR.AE_OVERFLOW	RUR_AE_OVERFLOW, RAE_OVERFLOW	Read state register AE_OVERFLOW
RUR.AE_SAR	RUR_AE_SAR, RAE_SAR	Read state register AE_SAR
RUR.AE_TABLESIZE	RUR_AE_TABLESIZE, RAE_TABLESIZE	Read state register AE_TABLESIZE
RUR.AE_FIRST_TS	RUR_AE_FIRST_TS, RAE_FIRST_TS	Read state register AE_FIRST_TS
RUR.AE_BITSUSED	RUR_AE_BITSUSED, RAE_BITSUSED	Read state register AE_BITSUSED
RUR.AE_BITPTR	RUR_AE_BITPTR, RAE_BITPTR	Read state register AE_BITPTR
RUR.AE_SEARCHDONE	RUR_AE_SEARCHDONE, RAE_SEARCHDONE	Read state register AE_SEARCHDONE
RUR.AE_NEXTOFFSET	RUR_AE_NEXTOFFSET, RAE_NEXTOFFSET	Read state register AE_NEXTOFFSET
WUR.AE_OVERFLOW	WUR_AE_OVERFLOW, WAE_OVERFLOW	Write state register AE_OVERFLOW
WUR.AE_SAR	WUR_AE_SAR, WAE_SAR	Write state register AE_SAR
WUR.AE_TABLESIZE	WUR_AE_TABLESIZE, WAE_TABLESIZE	Write state register AE_TABLESIZE
WUR.AE_FIRST_TS	WUR_AE_FIRST_TS, WAE_FIRST_TS	Write state register AE_FIRST_TS

Table 2-3 State Register Access Instructions

Instruction	Intrinsic	Description
WUR.AE_BITSUSED	WUR_AE_BITSUSED, WAE_BITSUSED	Write state register AE_BITSUSED
WUR.AE_BITPTR	WUR_AE_BITPTR, WAE_BITPTR	Write state register AE_BITPTR
WUR.AE_SEARCHDONE	WUR_AE_SEARCHDONE, WAE_SEARCHDONE	Write state register AE_SEARCHDONE
WUR.AE_NEXTOFFSET	WUR_AE_NEXTOFFSET, WAE_NEXTOFFSET	Write state register AE_NEXTOFFSET

Table 2-3 State Register Access Instructions

在本指南的操作说明中，每个助记符都列出了程序集语法，其中显示了其操作数的占位符。操作数的寄存器文件由占位符隐含，如表 2-4 所示。

Table 2-4 Operand Register Types

Placeholder	Register file	Legal values	Example
a, ah, al, a0, a1, ax	AR	a0 – a15	a3
p, p0, p1	AE_PR	aep0 – aep7	aep2
q, q0, q1, qh, ql	AE_QR	aeq0 – aeq3	aeq1
b	BR	b0 – b15	b3
bhl	BR2	b0 – b14 (even)	b0

Table 2-5 Operand Immediate Types

Placeholder	Value Range	Stride
i16	-16..14	2
i32	-32..28	4
i64	-64..56	8
i	Operation-dependent	1

每个操作说明都附有可以合法发出该操作的插槽的名称。第2.17节提供了可以在Inst、slot0和slot1中发出的所有HiFi 2/EP指令的摘要，以及可以在slot0中发出的所有核心指令。

每个操作说明还使用 C 语法进行批注，显示操作的内部名称和原型。有关使用 C 数据类型和内函数对 HiFi 2/EP DSP 进行编程的讨论包含在第 3 节中。

2.1 HiFi2/EP架构行为

HiFi 2/EP 架构在累加器数据路径和寄存器文件中包括保护位，以避免 ALU 和 MAC 操作溢出。这意味着此计算机上的典型数据流：

- 以较低的精度加载数据
- 使用保护位以更高的精度进行计算
- 以较低的精度存储结果

例如，通过将 24 位数据加载到操作数寄存器中，每个 24x24 位乘法在 56 位寄存器中产生 48 位数据，允许 8 个保护位进行累加。

这个是有意义的，用C语言通用的计算理解就像int型数据输入，然后里面的计算部分使用float类型计算，然后计算之后的数据输出成int型。这样的话计算部分的精度损失小，结果就更符合理论结果。

许多操作有饱和和非饱和版本。对于饱和操作，每当操作饱和时，都会设置AE_OVERFLOW状态。

2.2 指令命名约定

所有 HiFi 2/EP DSP 操作助记符都以字符串AE_开头，以避免与任何其他名称空间发生冲突。

在AE_前缀之后，每个助记符都有一个由一个或多个字符组成的字符串，表示操作类型，例如加载、移位、添加等。例如，AE_L 是表示 DSP 负载的前缀。

接下来，通常会指示包含操作的主要操作数或结果的寄存器文件。例如，AE_LP 是表示将值加载到寄存器 AE_PR的前缀。

通常，助记符中的寄存器文件指示器伴随着一个数字部分，作为相关对象大小的提醒。例如，AE_LP16 是表示 16 位数据项的 DSP 负载的前缀。

每个操作助记符的其余部分通常包括对操作详细信息各个方面的提醒。请参阅表 2-6。

Table 2-6 Operation Mnemonics

Mnemonic	Meaning
ASYM	Denotes asymmetric rounding (表示不对称舍入)
B	Denotes that the operation computes a Boolean result in addition to its numeric result(s) (表示该操作除了计算其数值结果外还计算布尔结果)
F	Denotes fractional arithmetic or the value False in a conditional move (表示分数算术或条件移动中的值“False”)
H and L	Combinations of H and L are used to refer to halves of registers in contexts where registers hold multiple values (H 和 L 的组合用于在寄存器保存多个值的上下文中指代寄存器的一半)

Mnemonic	Meaning
I	Denotes use of an immediate operand (表示使用即时操作数)
S	Denotes saturating arithmetic, signed arithmetic, or use of the AE_SAR state register as a shift amount, depending on context (表示饱和算术、有符号算术或使用AE_SAR状态寄存器作为移位量，具体取决于上下文)
SYM	Denotes symmetric rounding (表示对称舍入)
T	Denotes the value True in a conditional move (表示条件移动中的值 True)
U	Denotes unsigned arithmetic or an address register update depending on context (表示无符号算术或地址寄存器更新，具体取决于上下文)
X	Denotes use of an index register in an address computation (表示在地址计算中使用索引寄存器)
X2	Denotes a two-way SIMD operation in contexts (e.g., AE_PR loads and stores) where scalar operations are also available (表示在同时提供标量操作的上下文(例如，AE_PR加载和存储)中的双向 SIMD 操作)

用得比较多的是H和L，在24bit数据的操作里面非常常见，一般分别指48bit寄存器AE_PR里面的高位以及低位。

2.3 定点值和定点算术

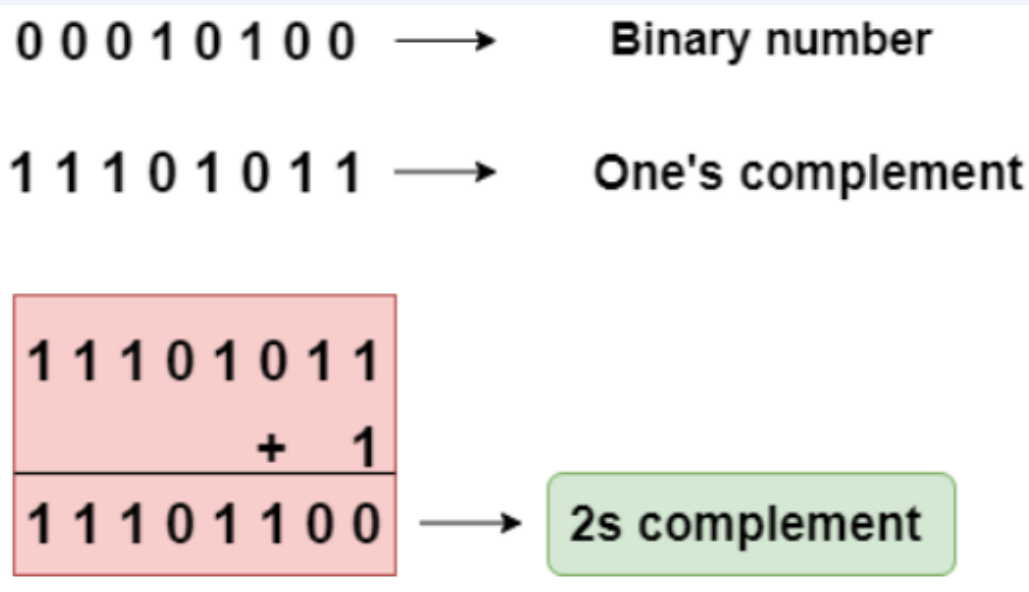
HiFi 2/EP DSP 包含实现定点运算的说明。本节介绍定点值的表示和解释，以及对定点值的一些操作。

2.3.1 定点值的表示

定点数据类型 `m.n` 包含一个符号位，小数点左侧包含一些位数的位数 $m-1$ （称为整数），小数点右侧包含一些位数的位 n （称为小数）。当表示为二进制值并存储到寄存器文件中时，**最低有效 n 位是小数部分，最高有效 m 位是表示为有符号 [二补数](#) 的整数部分**。如果二进制值被解释为 $2s$ 补码有符号整数，则从二进制值转换为定点数需要将整数除以 2^n 。

2s补码（二补数）是什么？

C中的2s补码由1s补码加一生成，1s补码是将原二进制文件0转换为1，1转换为0，2s补码就是在此基础上再加二进制1生成。后面会进行举例介绍。



2s补码形成过程

因此，例如，24位 1.23 数字0.5表示为0x400000。

	Signed Integer (1 bit) (整数部分)	Fractional (23 bits) (小数部分)
二进制	0	100 0000 0000 0000 0000 0000
十六进制	0x0	0x40 0000

这里的24bit (位) 数据的 $m.n = 1.23$ 。按照上面的说明，m包含一个符号位和m-1个整数位，n包含n个小数位，即上面的24位数据m=1、n=23，所以m部分除去一个符号位，剩0个整数位，n部分有23个小数位。所以上面表格里Signed Integer部分只有1bit，然后数值为0代表是正数（符号位），Fractional部分有23bit，下面说一下二进制的数怎么得出来的：

1. 因为小数部分有23位，所以将小数部分（0.5）乘以 2^{23} 取整，得出数值为4194304，注意此时的进制是10进制的，需要转为二进制，转换之后为100 0000 0000 0000 0000 0000，转换之后的二进制数值个数应该是23个。
2. 然后整数和符号位部分是一个0，将符号位、整数位和小数位concat起来，就变成了 0100 0000 0000 0000 0000 0000，然后转换成16进制方便查看就是 0x400000。每4个二进制合成一个16进制，所以16进制的位数有6位。

56 位 9.47 数字 -1.5 表示为 $(-2 + 0.5 = 0xff\ 4000\ 0000\ 0000)$

	Signed Integer (9 bit) (整数部分)	Fractional (47 bits) (小数部分)
二进制	1 1111 1110	100 0000 0000 0000 0000 0000 0000 0000 0000 0000
十六进制	0x1fe	0x4000 0000 0000

上面的56bit数字例子和最开始的24bit数字有以下不同：

1. 加入了整数部分
2. 表示数字是负数

下面对实现过程进行解析：

1. 首先需要清楚小数部分是没有符号位的，所以小数部分必须要是正数。上面的方法是 带符号整数+小数= 原本数值 $(-2+0.5=-1.5)$
2. 前面有说“**最高有效 m 位**是表示为有符号 [二补数](#)的**整数部分**”，所以整数部分并不是原本数值的二进制，而是原本数值的二补数。因为整数是负数。所以符号位为1，整数部分除了符号位还有8个整数位（这个应该不难理解），数字2的原本二进制是0000 0010，1s补码是1111 1101，2s补码是**1111 1110**，此时这个才是整数位的正常表示值，2s补码即为二补码。
3. 小数部分和之前是一样的，将小数部分（0.5）乘以 2^{47} 之后取整，得出数值为70368744177664，将其转为二进制为100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000，转换之后的二进制数值个数是47个。
4. 然后将整数部分和小数部分concat起来，变成1 1111 1110 100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000，然后从左边开始每四个合成一个16进制数值，那么最终得出的16进制数值为0xff 4000 0000 0000。

上面所说的小数部分乘以 2^n 取整，实际上是取四舍五入的整，这个判断是从网上看到的。[知 参考链接1](#)

为什么正数的补码（二补数）是其本身？[知 参考链接1](#)

HiFi 2/EP 分数指令在 [1.15](#)、[1.23](#)、[1.31](#) 和 [9.47](#) 上使用分数运算，详细描述如下。

- [1.15](#)：16 位定点数据类型，具有 1 个符号位和小数点右侧的 15 位。最大的正值0x7fff解释为 $1.0 - 2^{-15}$ 。最小的负值0x8000解释为 -1.0。值 0 被解释为 0.0。
- [1.23](#)：24 位定点数据类型，具有 1 个符号位和小数点右侧的 23 位。最大的正值0x7f ffff解释为 $1.0 - 2^{-23}$ 。最小的负值0x800000解释为 -1.0。值 0 被解释为 0.0。
- [1.31](#)：32 位定点数据类型，具有 1 个符号位和小数点右侧的 31 位。最大的正值0x7fff ffff解释为 $1.0 - 2^{-31}$ 。最小的负值0x80000000解释为 -1.0。值 0 被解释为 0.0。
- [9.47](#)：56 位定点数据类型，具有 9 位整数和小数点右侧的 47 位。最大的正值0x7f ffff ffff ffff解释为 $256.0 - 2^{-47}$ 。最小的负值 0x80 0000 0000 0000 解释为 -256.0。值 0 被解释为 0.0。

2.3.2 具有定点值的算术

当将定点数 $m0.n0 * m1.n1$ 与标准有符号整数乘数相乘时，倍数的自然结果将是 [m.n](#) 数据类型，其中 $n = n0+n1$ 和 $m = m0+m1$ 。因此，将 [1.23](#) 类型变量乘以 [1.23](#) 类型变量会生成 [2.46](#) 类型变量。由于HiFi 2 / EP支持 [9.47](#) 数据类型，因此定点乘法指令将 [2.46](#) 结果向左移动1位，然后符号扩展将其扩展7位。

向左移动1位是什么意思？

个人感觉是先把2.46变成1.47，然后在变成9.47。

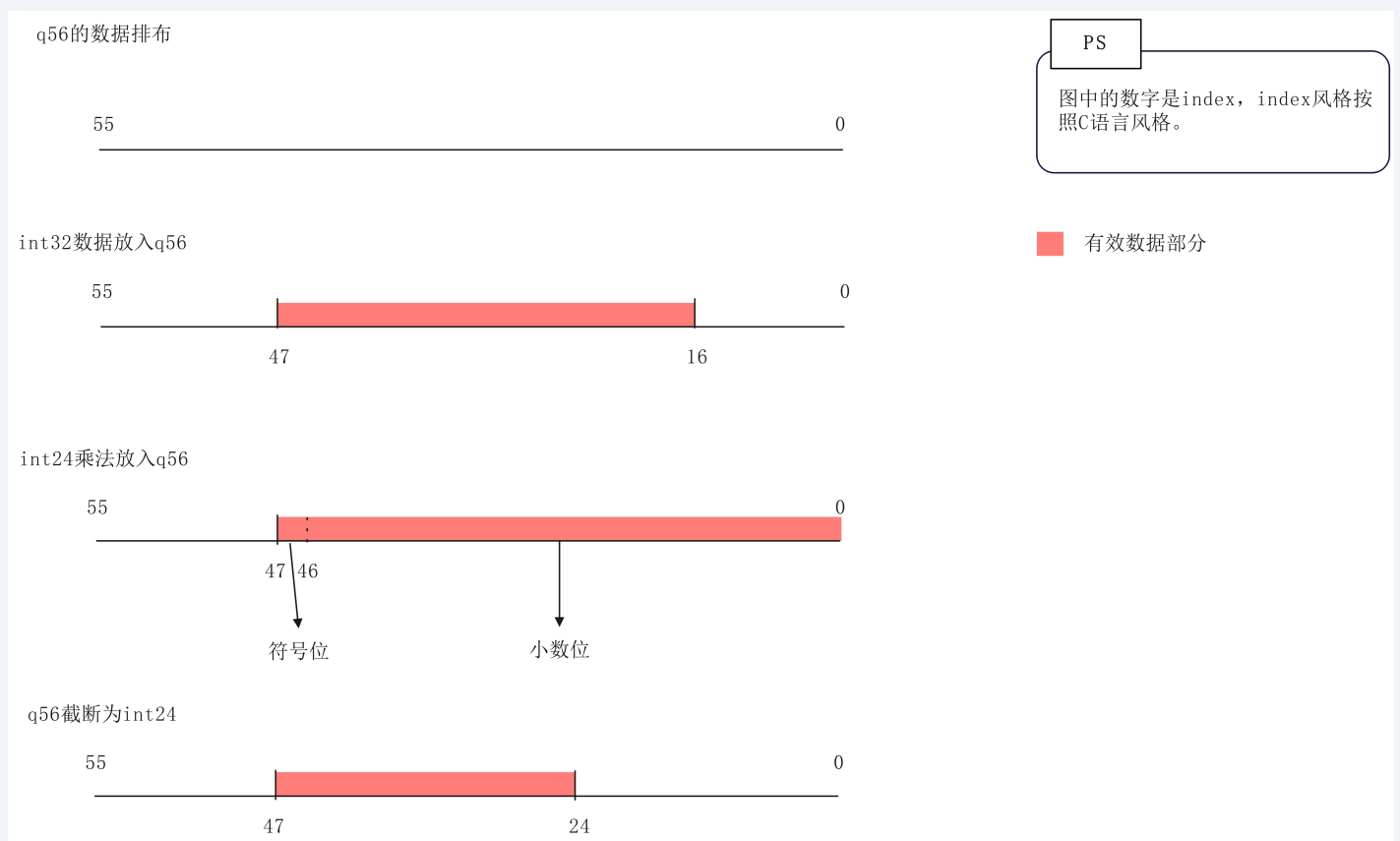
20221214更新

实际上发现得出的结果是比直接相乘大两倍的，比如我使用两个int16的 (1.15) 定点数值，一个0.5，一个0.3，定点化之后的十进制值为16384和9830，将两者相乘，得出数值为161054720，即 `AE_MULP24S_LL` 函数得出的结果，然后转成浮点发现是0.0749969482421875，显然是不对的；将两者相乘之后再乘2，得出数值为322109440，即 `AE_MULFP24S_LL` 函数得出的结果，转成浮点之后是0.149993896484375，这个数值就是正确的。所以可见两个定点数值相乘之后的数值确实需要乘2（即左移1位）。

这个也从侧面说明整数和定点分数的乘法使用的函数是不同的。

20230113更新

今天再更新一遍思路，因为在之前的时候把数据排布位置搞混了，导致一直无法很好的理解，现在把我进一步的理解描述一下



我比较不理解的部分是乘法之后的数据存储，因为q56是56位的，但是24位乘法之后就48位，那么肯定有一些位是用不到的，另一个不理解的就是从q56中提取32位数据，比如我乘法之后提取数据，那么如何才能确保数据是对的呢？

上图展示了q56的数据表示方法，0-55代表二进制位的位置索引，这样的排布方式我是看了[知 \(21 封私信 / 80 条消息\) 这个求指数函数exp\(\)的快速近似方法的原理是什么? - 知乎 \(zhihu.com\)](#)中一个回答画的图像参考的。当使用AE_LQ32F_I放置32位数据时候，根据函数描述就是**最高位（符号位）放到47索引那里，最低位（小数位）放到16索引那里**。当使用AE_MULP24S_LL进行24位乘法的时候同样的**最高位（符号位）放到47索引，最低位（小数位）放到0索引那里**。当使用AE_TRUNCP24Q48进行数据截断的时候，**符号位就是47索引，小数位就是24索引**，即截断只是把小数位的数目减少而已。

按照上面的理解，无论是放置数据，还是截取，都可以符合函数介绍里面描述。

还是这个其实还是解决不了向左移动一位的意思，除非数据的排布不是按照上面描述，而是符号位在低位，小数位在高位。向左移动这个只在使用定点乘法的时候才生效，如果使用整数乘法的时候是没有向左移动这个操作的，因此如果使用整数乘法代替定点乘法的话，那么就不用考虑向左移动这个情况。

HiFi 2/EP包含饱和和非饱和指令。用非饱和指令溢出提供的保护位是程序错误，将导致结果环绕。对于饱和操作，处理器还将设置溢出状态，稍后可以通过编程方式检查该状态。在下面的指令描述中，我们明确说明操作是否饱和。

2.3.3 其他定点表示

程序员可以自由使用上面列出的定点表示以外的定点表示。大多数HiFi 2/EP 操作独立于定点表示；例如，定点加法等效于整数加法。即使对于乘法，乘法指令也与任何期望结果左移一位的表示兼容。因此，**如果输入数据实际上是 2.22 数据类型而不是 1.23 数据类型，则 24 位定点乘法指令将正确生成 11.45 类型变量，作为两个 2.22 类型操作数相乘的结果**。程序员只是负责知道哪些变量中是什么类型的数据，如果需要手动转换，程序员可以随时使用移位指令。

2.4 VLIW插槽和格式

HiFi 2/EP可以使用Xtensa LX FLIX（VLIW）技术在单个64位指令束中发出两个操作。第一个插槽包含所有HiFi 2 / EP加载/存储指令、选择、56位移位操作和一些转换操作。第二个插槽包含所有乘法、DSP ALU、24 位移位和一些转换操作。**slot0 操作的子集以及所有比特流操作都以称为 Inst 的单个问题 24 位格式提供**。当不可能（或有益）将相关操作与可以放入第二个插槽的操作捆绑在一起时，编译器将自动使用 24 位格式。核心 Xtensa 操作的子集也可用于第一个 VLIW 插槽，允许 DSP 操作和核心 Xtensa 操作之间实现某种并行性。

仅在HiFi EP上，选择操作也可在第二个插槽中使用。如第4章所示，这在FIR和类似算法上提供了额外的性能。

在优化HiFi 2 / EP的代码时，了解插槽非常重要。通常，循环受到只能进入一个插槽或另一个插槽的操作的限制。例如，每个周期永远不可能发出多个乘法或多个（可能的 SIMD）加载或存储。如果循环受到一个槽中的操作的限制，则尝试优化另一个槽中的操作是没有意义的。

Inst 插槽中提供的所有 HiFi 2 指令与 MAC16 选件指令（op0=4'b0100）共享操作码空间（但不重叠）。

HiFi 2 slot0 和 slot1 采用 64 位指令格式，具有以下编码定义：

在下面的函数介绍中函数的名称后面使用 [插槽1 插槽2 插槽3 ...] 表示函数使用的插槽。inst插槽在图片中没有看到位置，但是很经常和slot0、slot1一起使用。

Little-Endian:

```
1 length l 64 { InstBuf[3:0] == 4'd15 }
2 format ae_format l { ae_slot0, ae_slot1 } {InstBuf[63:54] == 10'b0 }
```


Big-Endian:

```
1 length l 64 { InstBuf[63:60] == 4'd15 }
2 format ae_format l { ae_slot0, ae_slot1 } {InstBuf[9:0] == 10'b0 }
```

2.5 加载和存储操作

每个加载或存储操作的助记符都有一个后缀，指示如何计算**有效地址**以及是否将其写回寄存器。表 2-6 中列出了后缀。

有效地址是什么？

 [有效地址_百度百科 \(baidu.com\)](#)


 [\(76条消息\) 什么是有效地址和逻辑地址_ComputerInBook的博客-CSDN博客_有效地址](#)

Table 2-6 Load/Store Operation Suffixes

Suffix	Definition	Description
C (EP_only)	circular	如果有效地址穿过循环缓冲区的末尾，则地址将更新为缓冲区的开头。
I	immediate	有效地址是一个基本AR寄存器，加上一个 即时值 ，并且不会写回。
IU	immediate with update	有效地址是基本AR寄存器，加上一个即时值，基本寄存器使用有效地址进
X	indexed	有效地址是基本 AR 寄存器加上索引 AR 寄存器值，不会写回。
XU	indexed with update	有效地址为基本AR寄存器，加上偏移AR寄存器值，基本寄存器用有效地址

除了循环操作外，请注意，所有这些操作都遵循“预增量”约定；也就是说，在每种情况下，**有效地址都是将即时或寄存器偏移量添加到基址寄存器内容的结果**。所有循环缓冲区操作都遵循“后增量”约定；也就是说，在每种情况下，有效地址都是基址，而更新的基址是通过用圆形环绕将寄存器偏移量添加到基址来形成的。

有效地址简单理解就是指针的指向的位置，非循环操作里面，先将基址加上即时值或者偏移量，再赋予到有效地址；循环操作里面则是有效地址就是基址，然后加上偏移量之后再更新基址，有效地址就不变了。

这样的话使用I或IU的时候有效地址就是加上即时值之后的了，比如 `AE_LP16F_I`，i16数值即为即时值，有效地址就是基址加上i16，然后根据函数的逻辑就是加载有效地址的数据。**所以有效地址和即时值或者偏移量有没有关系是非常重要的**。基址就是指定的`ae_p16s * a`。

在每种情况下，地址都必须与加载或存储的内存对象的大小对齐。每个操作系列都注明了所需的地址对齐方式（以字节为单位）。

2.5.1 加载操作

```
1 AE_LP16F.I (.IU)    p, a, i16    [ slot0, Inst ]
2 AE_LP16F.X (.XU)    p, a, ax     [ slot0, Inst ]
```

所需对齐方式：2 字节 (bytes)

[知 bit \(比特\)和 Byte \(字节\) 的关系？ - 知乎 \(zhihu.com\)](#)
bit代表二进制中的一个数位，即“0”或“1”。bytes和bit的关系是 1bytes=8bit。

这些操作是为了满足处理16位数据的需求而提供的，如ITU G.7xx和3GPP AMR语音编解码器。这些操作从有效地址 (a + i16) 或 (a + ax) 加载一个 16 位值，并将左对齐的值（右侧用零填充）复制到AE_PR寄存器 p 的两个条目中的每一个。目的是内存中的值表示一个 16 位 (1.15) 分数，该分数作为 24 位 (1.23) 分数复制到AE_PR寄存器的两个条目中。

这里使用定点值，先加载一个16位值，16位二进制，比如1111 0000 1010 0101，使用左对齐的方式放到AE_PR寄存器p中的每一个，AE_PR寄存器p长度是48bit，有H部分和L部分，然后前面的16bit值会先变成24bit值，即1111 0000 1010 0101 0000 0000，然后**放置到AE_PR寄存器p的H部分和L部分，即H部分和L部分的数值是相同的**。

即时值i16的取值范围参照表2-5是-16到14，然后取值间隔是2，即取值范围是[-16, -14, -12...10, 12, 14]，这个单位是bit。

可以看到AE_LP16F字段中最后添加了F字符，通过查看表2-6可以发现F字符表示该函数用于分式算术，因此该函数的目的是加载一个分数。

C 语法:

C

```
1 ae_p24x2s AE_LP16F_I (const ae_p16s * a, immediate i16);
2 void AE_LP16F_IU (ae_p24x2s p /*out*/, const ae_p16s * a /*inout*/, immediate i16);
3 ae_p24x2s AE_LP16F_X (const ae_p16s * a, unsigned ax);
4 void AE_LP16F_XU (ae_p24x2s p /*out*/, const ae_p16s * a /*inout*/, unsigned ax);
```

C

```
1 AE_LP24.I (.IU)      p, a, i32    [ slot0, Inst ]
2 AE_LP24.X (.XU)      p, a, ax     [ slot0, Inst ]
```

所需对齐方式: 4 字节

这些操作从有效地址 (a + i32) 或 (a + ax) 处的 32 位字的最低有效 24 位加载 24 位值, 并将该值复制到寄存器 p AE_PR的两个条目中。

这个是用来读取32位数字的, 比如int32。不能用来读取16位的数值, 因为读取逻辑不一样可能会发生错误。

即时值取值范围同样参考表2-5。

C 语法:

C

```
1 ae_p24x2s AE_LP24_I (const ae_p24s * a, immediate i32);
2 void AE_LP24_IU (ae_p24x2s p /*out*/, const ae_p24s * a /*inout*/, immediate i32);
3 ae_p24x2s AE_LP24_X (const ae_p24s * a, unsigned ax);
4 void AE_LP24_XU (ae_p24x2s p /*out*/, const ae_p24s * a /*inout*/, unsigned ax);
```

C

```
1 AE_LP24F.I (.IU)      p, a, i32    [ slot0, Inst ]
2 AE_LP24F.X (.XU)      p, a, ax     [ slot0, Inst ]
```

所需对齐方式: 4 字节

这些操作从有效地址 (a + i32) 或 (a + ax) 处的 32 位字中最高的 24 位加载 24 位值, 并将该值复制到寄存器 p AE_PR 的两个条目中。目的是内存中的值表示一个 32 位 (1.31) 分数, 该分数被截断并复制到 p 的两个条目中, 作为 24 位 (1.23) 分数。

这个函数和前面的AE_LP16F一样, 只是加载的是24bit值而已。

C 语法:

```
C
1 ae_p24x2s AE_LP24F_I (const ae_p24f * a, immediate i32);
2 void AE_LP24F_IU (ae_p24x2s p /*out*/, const ae_p24f * a /*inout*/, immediate i32);
3 ae_p24x2s AE_LP24F_X (const ae_p24f * a, unsigned ax);
4 void AE_LP24F_XU (ae_p24x2s p /*out*/, const ae_p24f * a /*inout*/, unsigned ax);
```

```
C
1 AE_LP16X2F.I (.IU) p, a, i32 [ slot0, Inst ]
2 AE_LP16X2F.X (.XU) p, a, ax [ slot0, Inst ]
```

所需对齐方式: 4 字节

这些操作是为了满足处理16位数据的需求而提供的, 如ITU G.7xx和3GPP AMR语音编解码器。这些操作从有效地址 (a + i32) 或 (a + ax) 的内存中加载**一对** 16 位值, 并将每个值左对齐 (右侧用零填充) 放入 AE_PR寄存器 p 的条目中。目的是内存中的值表示 16 位 (1.15) 分数, 这些分数作为 24 位 (1.23) 分数放置在 p 的两个条目中。

和前面的AE_LP16F差不多, 但是是加载一对数据, 所以的话AE_PR寄存器 p 中的H和L是不一样的。

C 语法:

```
C
1 ae_p24x2s AE_LP16X2F_I (const ae_p16x2s * a, immediate i32);
2 void AE_LP16X2F_IU (ae_p24x2s p /*out*/, const ae_p16x2s * a /*inout*/, immediate i32);
3 ae_p24x2s AE_LP16X2F_X (const ae_p16x2s * a, unsigned ax);
4 void AE_LP16X2F_XU (ae_p24x2s p /*out*/, const ae_p16x2s * a /*inout*/, unsigned ax);
```


C

```
1 AE_LP24X2.I (.IU)  p, a, i64  [ slot0, Inst ]
2 AE_LP24X2.X (.XU)  p, a, ax   [ slot0, Inst ]
```

所需对齐方式：8 字节

这些操作将一对 24 位值加载到AE_PR寄存器 p 中，每个值来自有效地址 (a + i64) 或 (a + ax) 处 64 位的 32 位一半中最低有效 24 位。

C 语法：

C

```
1 ae_p24x2s AE_LP24X2_I (const ae_p24x2s * a, immediate i64);
2 void AE_LP24X2_IU (ae_p24x2s p /*out*/, const ae_p24x2s * a /*inout*/, immediate i64);
3 ae_p24x2s AE_LP24X2_X (const ae_p24x2s * a, unsigned ax);
4 void AE_LP24X2_XU (ae_p24x2s p /*out*/, const ae_p24x2s * a /*inout*/, unsigned ax);
```

C

```
1 AE_LP24X2F.I (.IU)  p, a, i64  [ slot0, Inst ]
2 AE_LP24X2F.X (.XU)  p, a, ax   [ slot0, Inst ]
```

所需对齐方式：8 字节

这些操作将一对 24 位值加载到AE_PR寄存器 p 中，每个值来自有效地址 (a + i64) 或 (a + ax) 处 64 位的 32 位中最重要的 24 位。目的是内存中的值表示 32 位 (1.31) 分数，这些分数被截断并作为 24 位 (1.23) 分数放置在AE_PR寄存器的两个条目中。

C 语法：

C

```
1 ae_p24x2s AE_LP24X2F_I (const ae_p24x2f * a, immediate i64);
2 void AE_LP24X2F_IU (ae_p24x2s p /*out*/, const ae_p24x2f * a /*inout*/, immediate i64);
3 ae_p24x2s AE_LP24X2F_X (const ae_p24x2f * a, unsigned ax);
4 void AE_LP24X2F_XU (ae_p24x2s p /*out*/, const ae_p24x2f * a /*inout*/, unsigned ax);
```

C

```
1 AE_LQ56.I (.IU)      q, a, i64    [ slot0, Inst ]
2 AE_LQ56.X (.XU)      q, a, ax     [ slot0, Inst ]
```

所需对齐方式：8 字节

这些操作将有效地址 (a + i64) 或 (a + ax) 处的 64 位中**最低有效 56 位**的 56 位值加载到AE_QR寄存器 q。

C 语法：

C

```
1 ae_q56s AE_LQ56_I (const ae_q56s * a, immediate i64);
2 void AE_LQ56_IU (ae_q56s q /*out*/, const ae_q56s * a /*inout*/, immediate i64);
3 ae_q56s AE_LQ56_X (const ae_q56s * a, unsigned ax);
4 void AE_LQ56_XU (ae_q56s q /*out*/, const ae_q56s * a /*inout*/, unsigned i64);
```

C

```
1 AE_LQ32F.I (.IU)      q, a, i32    [ slot0, Inst ]
2 AE_LQ32F.X (.XU)      q, a, ax     [ slot0, Inst ]
```

所需对齐方式：4 字节

这些操作将 32 位 (1.31) 值从有效地址 (a + i32) 或 (a + ax) 加载到 56 位AE_QR寄存器的 **16 到 47 位** 中。该值以符号方式扩展到 56 位AE_QR寄存器 q 的前 8 位，q 的下 16 位用零填充。

前8位就是高8位，下 16 位就是低16位，具体高低位的描述可以查看[👉高位字节、低位字节 - 大胖儿在努力 - 博客园 \(cnblogs.com\)](http://cnblogs.com)

相当于 (1.31) -> (1.47) 。原因可以看[这里](#)。

C 语法：

C

```
1 ae_q56s AE_LQ32F_I (const ae_q32s * a, immediate i32);
2 void AE_LQ32F_IU (ae_q56s q /*out*/, const ae_q32s * a /*inout*/, immediate i32);
3 ae_q56s AE_LQ32F_X (const ae_q32s * a, unsigned ax);
4 void AE_LQ32F_XU (ae_q56s q /*out*/, const ae_q32s * a /*inout*/, unsigned i32);
```

2.5.2 存放操作

C

```
1 AE_SP16X2F.I (.IU)  p, a, i32    [ slot0, Inst ]
2 AE_SP16X2F.X (.XU)  p, a, ax     [ slot0, Inst ]
```

所需对齐方式：4 字节

这些操作存储一对 16 位值，这些值来自AE_PR寄存器 p 中**每个 24 位元素的最高有效 16 位**。有效地址为 (a + i32) 或 (a + ax) 。

因为16bit数据放到AE_PR寄存器p时候是左对齐的，所以从AE_PR寄存器拿的时候自然也是拿高位的数值才匹配。具体可以看AE_LP16F定义。

需要注意的是这里是拿一对，所以如果H和L数值不一样的话那拿出来的一对数值也不一样。

C 语法：


C

```
1 void AE_SP16X2F_I (ae_p24x2s p, ae_p16x2s * a, immediate i32);
2 void AE_SP16X2F_IU (ae_p24x2s p, ae_p16x2s * a /*inout*/, immediate i32);
3 void AE_SP16X2F_X (ae_p24x2s p, ae_p16x2s * a, unsigned ax);
4 void AE_SP16X2F_XU (ae_p24x2s p, ae_p16x2s * a /*inout*/, unsigned ax);
```

C

```
1 AE_SP24X2S.I (.IU)  p, a, i64    [ slot0, Inst ]
2 AE_SP24X2S.X (.XU)  p, a, ax     [ slot0, Inst ]
```

所需对齐方式：8 字节

这些操作存储来自AE_PR寄存器 p 的一对 24 位值，每个值符号扩展到 32 位，并放置在有效地址 (a + i64) 或 (a + ax) 的 64 位中的一半。

C 语法：

C

```
1 void AE_SP24X2S_I (ae_p24x2s p, ae_p24x2s * a, immediate i64);
2 void AE_SP24X2S_IU (ae_p24x2s p, ae_p24x2s * a /*inout*/, immediate i64);
3 void AE_SP24X2S_X (ae_p24x2s p, ae_p24x2s * a, unsigned ax);
4 void AE_SP24X2S_XU (ae_p24x2s p, ae_p24x2s * a /*inout*/, unsigned ax);
```

C

```
1 AE_SP24X2F.I (.IU)  p, a, i64    [ slot0, Inst ]
2 AE_SP24X2F.X (.XU)  p, a, ax     [ slot0, Inst ]
```

所需对齐方式：8 字节

这些操作存储来自AE_PR寄存器 p 的一对 24 位值，每个值在右侧填充二进制数值“0”到 32 位，并放置在有效地址 (a + i64) 或 (a + ax) 的 64 位中的一半。目的是寄存器 p 中的值表示填充到 32 位 (1.31) 内存表示形式的 24 位 (1.23) 分数。

C 语法：

C

```
1 void AE_SP24X2F_I (ae_p24x2s p, ae_p24x2f * a, immediate i64);
2 void AE_SP24X2F_IU (ae_p24x2s p, ae_p24x2f * a /*inout*/, immediate i64);
3 void AE_SP24X2F_X (ae_p24x2s p, ae_p24x2f * a, unsigned ax);
4 void AE_SP24X2F_XU (ae_p24x2s p, ae_p24x2f * a /*inout*/, unsigned ax);
```

C

```
1 AE_SP16F.L.I (.IU)  p, a, i16    [ slot0, Inst ]
2 AE_SP16F.L.X (.XU)  p, a, ax     [ slot0, Inst ]
```

所需对齐方式：2 字节

这些操作将AE_PR寄存器 p 的 24 位 **L 元素的前 16 位**存储到有效地址 (a + i16) 或 (a + ax) 处的 16 位。

这个是只拿一个16bit的数值，和前面的拿一对不同，需要注意一下。

C 语法：

C

```
1 void AE_SP16F_L_I (ae_p24x2s p, ae_p16s * a, immediate i16);
2 void AE_SP16F_L_IU (ae_p24x2s p, ae_p16s * a /*inout*/, immediate i16);
3 void AE_SP16F_L_X (ae_p24x2s p, ae_p16s * a, unsigned ax);
4 void AE_SP16F_L_XU (ae_p24x2s p, ae_p16s * a /*inout*/, unsigned ax);
```

C

```
1 AE_SP24S.L.I (.IU)  p, a, i32    [ slot0, Inst ]
2 AE_SP24S.L.X (.XU)  p, a, ax     [ slot0, Inst ]
```

所需对齐方式：4 字节

这些操作对AE_PR寄存器 p 的 24 位 L 元素进行符号扩展和存储到有效地址 (a + i32) 或 (a + ax) 处的 32 位。

C 语法：

C

```
1 void AE_SP24S_L_I (ae_p24x2s p, ae_p24s * a, immediate i32);
2 void AE_SP24S_L_IU (ae_p24x2s p, ae_p24s * a /*inout*/, immediate i32);
3 void AE_SP24S_L_X (ae_p24x2s p, ae_p24s * a, unsigned ax);
4 void AE_SP24S_L_XU (ae_p24x2s p, ae_p24s * a /*inout*/, unsigned ax);
```

C

```
1 AE_SP24F.L.I (.IU)  p, a, i32    [ slot0, Inst ]
2 AE_SP24F.L.X (.XU)  p, a, ax     [ slot0, Inst ]
```

所需对齐方式：4 字节

这些操作将AE_PR寄存器 p 的 24 位 L 元素（右侧用零填充）存储到有效地址 (a + i32) 或 (a + ax) 处的 32 位。

这个可不可以理解成是 (1.23) 变成 (1.31) ？

C 语法：

C

```
1 void AE_SP24F_L_I (ae_p24x2s p, ae_p24f * a, immediate i32);
2 void AE_SP24F_L_IU (ae_p24x2s p, ae_p24f * a /*inout*/, immediate i32);
3 void AE_SP24F_L_X (ae_p24x2s p, ae_p24f * a, unsigned ax);
4 void AE_SP24F_L_XU (ae_p24x2s p, ae_p24f * a /*inout*/, unsigned ax);
```

C

```
1 AE_SQ56S.I (.IU)    q, a, i64    [ slot0, Inst ]
2 AE_SQ56S.X (.XU)    q, a, ax     [ slot0, Inst ]
```

所需对齐方式：8 字节

这些操作对AE_QR寄存器 q 的 56 位进行符号扩展和存储到有效地址 (a + i64) 或 (a + ax) 处的 64 位。

C 语法：

C

```
1 void AE_SQ56S_I (ae_q56s q, ae_q56s * a, immediate i64);
2 void AE_SQ56S_IU (ae_q56s q, ae_q56s * a /*inout*/, immediate i64);
3 void AE_SQ56S_X (ae_q56s q, ae_q56s * a, unsigned ax);
4 void AE_SQ56S_XU (ae_q56s q, ae_q56s * a /*inout*/, unsigned ax);
```

C

```
1 AE_SQ32F.I (.IU)    q, a, i32    [ slot0, Inst ]
2 AE_SQ32F.X (.XU)    q, a, ax     [ slot0, Inst ]
```

所需对齐方式：4 字节

这些操作将 32 位 (1.31) 值从 56 AE_QR位的位 16 到 47 存储到有效地址 (a + i32) 或 (a + ax) 。

如果是定点分数的话，直接取16到47位的数据应该不行吧？为什么实际上这个是可行的呢？比如下面的这种情况两个(1.23)数值相乘的情况

这样应该是从第9位开始取吧？

相当于 (1.47) -> (1.31)

C 语法：

```
C
1 void AE_SQ32F_I (ae_q56s q, ae_q32s * a, immediate i32);
2 void AE_SQ32F_IU (ae_q56s q, ae_q32s * a /*inout*/, immediate i32);
3 void AE_SQ32F_X (ae_q56s q, ae_q32s * a, unsigned ax);
4 void AE_SQ32F_XU (ae_q56s q, ae_q32s * a /*inout*/, unsigned ax);
```

2.6 循环缓冲区加载/存储操作（EP only）

每个循环缓冲区加载或存储操作的助记符都有一个 .C 后缀指示操作正在使用循环缓冲区寻址模式。循环缓冲区边界通过两种 32 位状态指定：

Table 2-7 Circular Buffer States

State	Description
AE_CBEGIN0	The start address of the circular buffer.（循环缓冲区的起始地址。）
AE_CEND0	The end address of circular buffer, i.e., the start address, plus the byte size of the buffer.（循环缓冲区的结束地址，即开始地址，加上缓冲区的字节大小。）

以下内部函数可用于从 C 中的循环缓冲区状态读取：

```
C
1 void * AE_GETCBEGIN0 (void);
2 void * AE_GETCEND0 (void);
```

以下内部函数可用于写入 C 中的循环缓冲区状态：

```
C
1 void AE_SETCBEGIN0 (const void * addr);
2 void AE_SETCEND0 (const void * addr);
```

所有循环缓冲区操作都遵循“后增量”约定，即在每种情况下，有效地址都是基址，而更新的基址是通过使用圆形环绕将寄存器偏移量添加到基址来形成的。这与遵循“预增量”约定的其他HiFi 2加载和存储不同。

地址增量以字节数指定，并且必须小于或等于缓冲区字节大小。增量可以是正的（缓冲区末尾的**环绕**），也可以是负的（缓冲区开头的环绕）。

这个缓冲区的环境不太明白什么意思

在每种情况下，基址都必须与加载或存储的内存对象的大小对齐。

下面是演示如何初始化和使用循环缓冲区的示例 C 代码片段。缓冲区用于在每个 32 位字的 24 MSB 中存储 24 位数据，从缓冲区的最后一个元素开始以负步幅开始。

```
C

1  /* Allocate the buffers. */
2  void *buf = malloc(buf_size);
3
4  /* Initialize the circular buffer boundaries. */
5  AE_SETCBEGIN0(buf);
6  AE_SETCEND0(buf + buf_size);
7
8  /* Point to the first element to be loaded/stored. */
9  ae_p24f *buf_ptr = (ae_p24f *) (buf + buf_size - sizeof(ae_p24f));
10 ...
11 for (...) {
12     ae_p24x2s p;
13     ...
14     AE_SP24F_C(p, buf_ptr, -sizeof(ae_p24f));
15     ...
16 }
```

```
C

1  AE_LP16F.C  p, a, ax  [ slot0 ]
2  AE_LP24F.C  p, a, ax  [ slot0 ]
3  AE_LP24.C   p, a, ax  [ slot0 ]
4  AE_LP16X2F.C  p, a, ax  [ slot0 ]
5  AE_LP24X2F.C  p, a, ax  [ slot0 ]
6  AE_LP24X2.C   p, a, ax  [ slot0 ]
7  AE_LQ32F.C  q, a, ax  [ slot0 ]
8  AE_LQ56.C   q, a, ax  [ slot0 ]
9  AE_SP16F.L.C  p, a, ax  [ slot0 ]
10 AE_SP24F.L.C  p, a, ax  [ slot0 ]
11 AE_SP24S.L.C  p, a, ax  [ slot0 ]
12 AE_SP16X2F.C  p, a, ax  [ slot0 ]
13 AE_SP24X2F.C  p, a, ax  [ slot0 ]
14 AE_SP24X2S.C  p, a, ax  [ slot0 ]
15 AE_SQ32F.C  q, a, ax  [ slot0 ]
16 AE_SQ56S.C  q, a, ax  [ slot0 ]
```

这些操作从有效地址a加载数据或将数据存储到有效地址a。

地址寄存器 a 更新如下：

```
C
1  a <- a+ax-CEND0+CBEGIN0, if (a+ax) ≥ CEND0 and ax is positive (wrap-around);
2  a <- a+ax+CEND0-CBEGIN0, if (a+ax) < CBEGIN0 and ax is negative (wrap-around);
3  a <- a+ax, otherwise (no circular buffer wrap-around).
```

请注意，将 CBEGIN0 和 CEND0 指定为 0 会导致跨越整个 32 位地址空间的循环缓冲区，即循环加载/存储操作将作为常规的增量后索引加载/存储操作运行。

有关每个操作的数据格式和对齐要求的详细说明，请参阅具有相同操作码助记符和 X 后缀的相应加载/存储操作。

C 语法：

```
C
1  void AE_LP16F_C (ae_p24x2s p /*out*/, const ae_p16s * a /*inout*/, int ax);
2  void AE_LP24F_C (ae_p24x2s p /*out*/, const ae_p24f * a /*inout*/, int ax);
3  void AE_LP24_C (ae_p24x2s p /*out*/, const ae_p24s * a /*inout*/, int ax);
4  void AE_LP16X2F_C (ae_p24x2s p /*out*/, const ae_p16x2s * a /*inout*/, int ax);
5  void AE_LP24X2F_C (ae_p24x2s p /*out*/, const ae_p24x2f * a /*inout*/, int ax);
6  void AE_LP24X2_C (ae_p24x2s p /*out*/, const ae_p24x2s * a /*inout*/, int ax);
7  void AE_LQ32F_C (ae_q56s q /*out*/, const ae_q32s * a /*inout*/, int ax);
8  void AE_LQ56_C (ae_q56s q /*out*/, const ae_q56s * a /*inout*/, int ax);
9  void AE_SP16F_L_C (ae_p24x2s p, ae_p16s * a /*inout*/, int ax);
10 void AE_SP24F_L_C (ae_p24x2s p, ae_p24f * a /*inout*/, int ax);
11 void AE_SP24S_L_C (ae_p24x2s p, ae_p24s * a /*inout*/, int ax);
12 void AE_SP16X2F_C (ae_p24x2s p, ae_p16x2s * a /*inout*/, int ax);
13 void AE_SP24X2F_C (ae_p24x2s p, ae_p24x2f * a /*inout*/, int ax);
14 void AE_SP24X2S_C (ae_p24x2s p, ae_p24x2s * a /*inout*/, int ax);
15 void AE_SQ32F_C (ae_q56s q, ae_q32s * a /*inout*/, int ax);
16 void AE_SQ56S_C (ae_q56s q, ae_q56s * a /*inout*/, int ax);
```

2.7 乘法和累加运算

乘法/累加运算形成了一个丰富的集合，可以沿多个维度划分为族。在一个维度上，它们分为做两次乘法的乘法和只做一次乘法的乘法。在另一个维度上，它们分为执行 24x24 位乘法的乘法和执行 32x16 位乘法的乘法。此外，HiFi 2/EP 配置选项支持 32x24 位乘法。在以下部分中，我们将首先将运算划分为乘法精度类别，在每个类别中，将首先描述单乘法组，然后描述双乘法组。

2.7.1 24x24 位乘法运算

24x24 位乘法运算的输入操作数是寄存器的AE_PR元素。每个AE_PR寄存器包含两个24位元件;对于乘法的每个AE_PR寄存操作数，必须选择两个元素中的一个作为乘法的输入。为此，每个乘法运算都有一个后缀，指示每个AE_PR操作数是选择 L 还是 H 元素。每个乘法/累加运算的结果都会进入AE_QR寄存器，该寄存器有时称为累加器。

有8个AE_PR寄存器，每个寄存器有两个元素：H和L。对于24x24位乘法，实际上是两个AE_PR寄存器之间的乘法，会各自从两个寄存器之间选择H或L作为乘法的输入。之前的使用一个寄存器H和L相乘的想法是错误的。

2.7.1.1 24x24 位和 16x16 位单乘法运算

在单乘运算中，每个乘法/累加运算族都有一个仅乘法变体、一个乘法/加法变体和一个乘法/减法变体，分别由 MUL、MULA 和 MULS 助记符子字符串表示。使用 MUL 变体时，累加器内容将被乘法结果覆盖。对于 MULA 变体，乘法的结果将累加到累加器内容中并写回累加器。对于 MULS 变体，乘法的结果将从累加器内容中减去并写回累加器。

我对这个进行了测试，具体的测试文档路径。里面的tmp2就是累加器。

```
C
1 AE_MULP24S.LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
2 AE_MULAP24S.LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
3 AE_MULSP24S.LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
```

24x24 至 56 位结果带符号整数 MAC，无饱和。关于相乘之后放置到56位的位置可以查看2.3.2 具有定点值的算术。

这里解释了为什么当时“HiFi2 入门探索（一）-20221205”文档中这里有个问题，就是我的数据类型定义是Unsigned类型，按理来说应该是没有负数的呀，怎么来的负数？得出tmp2可以是负数。其实可能是从根源上已经改变了指针所指向数值的类型（U32->int32）。

20230203更新

其实uint32和int32之间时可以相互转换的，编译器运算的时候会自动转换，想要正确显示的话需要使用正确的显示符号，后面的文档会更新。 #许愿池

C 语法：

C

```
1 ae_q56s AE_MULP24S_LL (ae_p24x2s p0, ae_p24x2s p1);
2 void AE_MULAP24S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
3 void AE_MULSP24S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
```

C

```
1 AE_MULFP24S_LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
2 AE_MULAFP24S_LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
3 AE_MULSFP24S_LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
```

24x24 位有符号定点分数 (1.23) [-1.0 .. 1.0) 到 56 位 (9.47) 结果 MAC 无饱和。

这个和上面的AE_MULP24S_LL的区别是什么？目前来看就只是数值2倍？

C 语法：

C

```
1 ae_q56s AE_MULFP24S_LL (ae_p24x2s p0, ae_p24x2s p1);
2 void AE_MULAFP24S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
3 void AE_MULSFP24S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
```

C

```
1 AE_MULAS56P24S_LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
2 AE_MULSS56P24S_LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
```

24x24 至 56 位结果带符号整数 MAC，具有 56 位累加器饱和度。

C 语法：

C

```
1 void AE_MULAS56P24S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
2 void AE_MULSS56P24S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
```

```
C
1 AE_MULAFS56P24S.LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
2 AE_MULSFS56P24S.LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
```

24x24 位有符号分数 (1.23) [-1.0 .. 1.0) 到 56 位 (9.47) MAC, 累加器饱和度为 56 位。

C 语法:

```
C
1 void AE_MULAFS56P24S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
2 void AE_MULSFS56P24S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
```

```
C
1 AE_MULFS32P16S.LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
2 AE_MULAFS32P16S.LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
3 AE_MULSFS32P16S.LL (.LH .HL .HH)    q, p0, p1    [ slot1 ]
```

16x16 位有符号分数 (1.15) [-1.0 .. 1.0) 到 32 位 (1.31) MAC, 具有 32 位乘积和累加器饱和度。请注意, 二进制点的位置分别相对于 1.23 和 9.47 值, 与 AE_LP16F/AE_SP16F 和 AE_LQ32F/AE_SQ32F 操作匹配的格式相同。

C 语法:

```
C
1 ae_q56s AE_MULFS32P16S_LL (ae_p24x2s p0, ae_p24x2s p1);
2 void AE_MULAFS32P16S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
3 void AE_MULSFS32P16S_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
```

2.7.1.2 24x24 位双乘法运算

每个双乘法运算等效于来自同一系列的两个连续单乘法运算 (它们之间可能有 AE_NEGQ56 运算) 并且具有相同的寄存器操作数。在双乘法运算中**仅表示非饱和计算**。

因为每个对偶乘法运算都进行两次乘法和累加计算，所以助记符具有相应的额外复杂性。名称为 MULZ 的操作对零进行累加；换句话说，累加器的初始内容被丢弃。那些没有 Z 的值根据累加器的初始内容累加。在可选的 Z 之后有两个字母表示加法或减法，一个代表两个乘法结果中的每一个。对应于这两个加/减指示符，有一个后缀 HL.LH 或 HH.LL（只有这两种写法），指示如何从元素 AE_PR 寄存器操作数中选择乘法的输入。在 HL.LH 的情况下，两个乘法结果是：第一个 AE_PR 操作数的 H 元素乘以第二个 AE_PR 操作数的 L 元素；第一个 AE_PR 操作数的 L 元素乘以第二个 AE_PR 操作数的 H 元素。对于 HH.LL，第一个乘法结果是两个 AE_PR 操作数的 H 元素的乘积，第二个乘法结果是两个 AE_PR 操作数的 L 元素的乘积。

以 AE_MULZAAP24S.HH.LL 举例，AE_MUL 和之前的一样，代表是乘法，后面的 Z 代表累加器初始值内容被丢弃，或者说到时候被覆盖，再后面的 AA 代表两次乘法都有累加，后面的 (F)P24S 代表寄存器使用 AE_PR、24 位、带符号(定点)整数类型，HH 和 LL 代表两次乘法的位置，第一个乘法的位置是两个 AE_PR 寄存器的高位和低位，第二个乘法的位置是两个 AE_PR 寄存器的低位和低位。（24 后面的 S 代表有符号）

这里有一个问题，两次乘法的话后面一次乘法应该是在前一次乘法的基础上进行数值操作的吧？因为累加器只能放一次乘法的结果，比如上面的例子，后面一次乘法应该是和前面一次乘法的结果进行累加吧？

```
C
1 AE_MULZAAP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
2 AE_MULZASP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
3 AE_MULZSAP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
4 AE_MULZSSP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
5 AE_MULAAP24S.HH.LL (.HL.LH)     q, p0, p1    [ slot1 ]
6 AE_MULASP24S.HH.LL (.HL.LH)     q, p0, p1    [ slot1 ]
7 AE_MULSAP24S.HH.LL (.HL.LH)     q, p0, p1    [ slot1 ]
8 AE_MULSSP24S.HH.LL (.HL.LH)     q, p0, p1    [ slot1 ]
```

24x24 至 56 位结果带符号整数 MAC，无饱和。

C 语法：

```
C
1 ae_q56s AE_MULZAAP24S_HH_LL (ae_p24x2s p0, ae_p24x2s p1);
2 void AE_MULAAP24S_HH_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);
```

```
C
1 AE_MULZAAFP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
2 AE_MULZASFP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
3 AE_MULZSAFP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
4 AE_MULZSSFP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
```

```

5 AE_MULAAFP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
6 AE_MULASFP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
7 AE_MULSAFP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]
8 AE_MULSSFP24S.HH.LL (.HL.LH)    q, p0, p1    [ slot1 ]

```

24x24 位有符号分数 (1.23) [-1.0 .. 1.0) 到 56 位 (9.47) MAC, 无饱和。例如, 如果 aep0 和 aep1 各自持有一个复数, 其中 H 元素中有实数部分, 则以下两条指令以分数算术计算两个数字的乘积:

```

C
1 AE_MULZASFP24S.HH.LL    aeq0, aep0, aep1    // real part
2 AE_MULZAFP24S.HL.LH     aeq1, aep0, aep1    // imaginary part

```

C 语法:

```

C
1 ae_q56s AE_MULZAFP24S_HH_LL (ae_p24x2s p0, ae_p24x2s p1);
2 void AE_MULAAFP24S_HH_LL (ae_q56s q /*inout*/, ae_p24x2s p0, ae_p24x2s p1);

```

2.7.2 32x16 位乘法运算

根据AE_PR和AE_QR寄存器文件的体系结构, 32x16 位乘法运算的输入操作数是异构的: 32 位输入来自 AE_QR, 16 位输入来自AE_PR。与 24 位乘法运算一样, AE_PR输入的相关元素在助记符中使用后缀 L 或 H 进行选择。结果要么写入, 要么累积到AE_QR寄存器。32x16 位乘法运算均不包括饱和算术。

2.7.2.1 32x16 位单乘法运算

```

C
1 AE_MULQ32SP16S.L (.H)    q, q0, p0    [ slot1 ]
2 AE_MULAQ32SP16S.L (.H)    q, q0, p0    [ slot1 ]
3 AE_MULSQ32SP16S.L (.H)    q, q0, p0    [ slot1 ]

```

32x16 至 56 位结果带符号整数 MAC, 无饱和。输出 q0×p0.[LH] 覆盖、累加或减去 q。

就是相乘之后的数值和原本累加器q上面的数值进行覆盖、累加、减去操作, 其中减去的操作是累加器原本的值减去相乘之后的数值, 覆盖是相乘之后的数值代替原本累加器的数值。

C 语法:

C

```
1 ae_q56s AE_MULQ32SP16S_L (ae_q56s q0, ae_p24x2s p0);
2 void AE_MULAQ32SP16S_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
3 void AE_MULSQ32SP16S_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
```

C

```
1 AE_MULQ32SP16U.L (.H) q, q0, p0 [ slot1 ]
2 AE_MULAQ32SP16U.L (.H) q, q0, p0 [ slot1 ]
3 AE_MULSQ32SP16U.L (.H) q, q0, p0 [ slot1 ]
```

32 位由 16 位**无符号**到 56 位结果整数 MAC 无饱和。输出 $q0 \times p0$. [LH] 覆盖、累加或减去 q 。

C 语法:

C

```
1 ae_q56s AE_MULQ32SP16U_L (ae_q56s q0, ae_p24x2s p0);
2 void AE_MULAQ32SP16U_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
3 void AE_MULSQ32SP16U_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
```

C

```
1 AE_MULFQ32SP16S.L (.H) q, q0, p0 [ slot1 ]
2 AE_MULAFQ32SP16S.L (.H) q, q0, p0 [ slot1 ]
3 AE_MULSFQ32SP16S.L (.H) q, q0, p0 [ slot1 ]
```

32x16 位有符号部分 (1.31 x 1.15) [-1.0 .. 1.0] 到 56 位 (9.47) MAC, 无饱和。输出 $q0 \times p0$. [LH] 覆盖、累加或减去 q 。

C 语法:

C

```
1 ae_q56s AE_MULFQ32SP16S_L (ae_q56s q0, ae_p24x2s p0);
2 void AE_MULAFQ32SP16S_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
3 void AE_MULSFQ32SP16S_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
```

C

```
1 AE_MULFQ32SP16U.L (.H) q, q0, p0 [ slot1 ]
2 AE_MULAFQ32SP16U.L (.H) q, q0, p0 [ slot1 ]
3 AE_MULSFQ32SP16U.L (.H) q, q0, p0 [ slot1 ]
```

32 位有符号定点分数 (1.31) [-1.0 .. 1.0) 乘以 16 位无符号分数 (0.16) [0 .. 1.0) 到 56 位 (9.47) MAC, 无饱和。输出 $q0 \times p0.[LH]$ 向左移动一个位置, 并覆盖、累加或减去 q 。

C 语法:

C

```
1 ae_q56s AE_MULFQ32SP16U_L (ae_q56s q0, ae_p24x2s p0);
2 void AE_MULAFQ32SP16U_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
3 void AE_MULSFQ32SP16U_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
```

2.7.2.2 32x16 位双乘法运算

每个双乘法运算等效于来自同一系列的两个连续单乘法运算 (它们之间可能有 AE_NEGQ56 运算) 并且具有相同的寄存器操作数。在双乘法运算中仅表示非饱和计算。

所有 32x16 位双乘法运算都对零进行累加, 并且它们的助记符都相应地以 AE_MULZ 开头。命名方案遵循 24x24 位乘法运算的模式。

C

```
1 AE_MULZAAQ32SP16S.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
2 AE_MULZASQ32SP16S.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
3 AE_MULZSAQ32SP16S.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
4 AE_MULZSSQ32SP16S.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
```

32x16 到 56 位结果有符号整数**乘法/加法**, 无饱和。该值 $\pm q0 \times p0.[LH] \pm q1 \times p1.[LH]$ 写入 q 。

这里对上面的式子进行介绍, 以 AE_MULZAAQ32SP16S.LL 举例

AE_MUL 代表乘法, 后面的 Z 代表累加器归零, 后面的 AA 代表双乘法都用加法, 即 $0 + q0 \times p0.[LH] + q1 \times p1.[LH]$ 。后面的 q32s 代表一个乘数的位置是累加器, 32 位输入, 带符号, 后面的 P16S 代表另一个乘数的位置是 AE_PR 寄存器, 16 位输入, 带符号, 后面的 LL 代表双乘法都取两个 AE_PR 寄存器的低位。

C 语法:

C

```
1 ae_q56s AE_MULZAAQ32SP16S_LL (ae_q56s q0, ae_p24x2s p0, ae_q56s q1, ae_p24x2s p1);
```

C

```
1 AE_MULZAAFQ32SP16S.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
2 AE_MULZASFQ32SP16S.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
3 AE_MULZSAFQ32SP16S.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
4 AE_MULZSSFQ32SP16S.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
```

32x16 位有符号分数 (1.31 x 1.15) [-1.0 .. 1.0) 到 56 位 (9.47) 乘法/加法而不饱和。该值 $\pm q0 \times p0.[LH] \pm q1 \times p1.[LH]$ 写入 q。

C 语法:

C

```
1 ae_q56s AE_MULZAAFQ32SP16S_LL (ae_q56s q0, ae_p24x2s p0, ae_q56s q1, ae_p24x2s p1);
```

C

```
1 AE_MULZAAQ32SP16U.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
2 AE_MULZASQ32SP16U.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
3 AE_MULZSAQ32SP16U.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
4 AE_MULZSSQ32SP16U.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
```

32 位由 16 位无符号到 56 位结果**整数**乘法/加法而不饱和。该值 $\pm q0 \times p0.[LH] \pm q1 \times p1.[LH]$ 写入 q。

和之前的区别是16位是无符号的。

C 语法:

C

```
1 ae_q56s AE_MULZAAQ32SP16U_LL (ae_q56s q0, ae_p24x2s p0, ae_q56s q1, ae_p24x2s p1);
```

```
C
1 AE_MULZAAFQ32SP16U.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
2 AE_MULZASFQ32SP16U.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
3 AE_MULZSAFQ32SP16U.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
4 AE_MULZSSFQ32SP16U.LL (.LH .HH) q, q0, p0, q1, p1 [ slot1 ]
```

32 位有符号分数 (1.31) x 16 位无符号**分数** (0.16) [0 .. 1.0) 到 56 位 (9.47) MAC, 无饱和。该值 $\pm q0 \times p0.[LH] \pm q1 \times p1.[LH]$ 写入 q。

这个和上面相比是定点的。

C 语法:

```
C
1 ae_q56s AE_MULZAAFQ32SP16U_LL (ae_q56s q0, ae_p24x2s p0, ae_q56s q1, ae_p24x2s p1);
```

2.7.3 32x24 位乘法运算 (EP only)

HiFi 2/EP 配置选项扩展了 HiFi 2 ISA, 支持 32x24 位乘法。根据 AE_PR 和 AE_QR 寄存器文件的体系结构, 32x24 位乘法的输入操作数是异构的: **32 位输入来自 AE_QR, 24 位输入来自 AE_PR**。AE_PR 输入的相关元素是使用助记符中的后缀 L 或 H 选择的。结果要么写入, 要么累积到 AE_QR 寄存器。32x24 位乘法运算均**不包括饱和算术或双乘法**。

2.7.3.1 32x24 位单乘法运算

```
C
1 AE_MULFQ32SP24S.L (.H) q, q0, p0 [ slot1 ]
2 AE_MULAFQ32SP24S.L (.H) q, q0, p0 [ slot1 ]
3 AE_MULSFQ32SP24S.L (.H) q, q0, p0 [ slot1 ]
```

32x24 位有符号部分 (1.31 x 1.23) [-1.0 .. 1.0) 到 50 位 (3.47) MAC, 无饱和。56 位 (2.54) 可能否定乘积 $\pm q0 \times p0.[LH]$ 被截断为 $-\infty$ 并符号扩展为 50 位 (3.47)。然后将乘积写入或添加到 q 的 50 LSBs 中, 最终的 50 位结果被符号扩展为 56 位 (9.47) 定点值。

可能否定乘积 $\pm q0 \times p0.[LH]$ 这个意思不是很理解? 是不是取反的意思?

C 语法:

C

```
1 ae_q56s AE_MULFQ32SP24S_L (ae_q56s q0, ae_p24x2s p0);
2 void AE_MULAFQ32SP24S_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
3 void AE_MULSFQ32SP24S_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
```

C

```
1 AE_MULRFQ32SP24S.L (.H)    q, q0, p0    [ slot1 ]
2 AE_MULARFQ32SP24S.L (.H)    q, q0, p0    [ slot1 ]
3 AE_MULSRFQ32SP24S.L (.H)    q, q0, p0    [ slot1 ]
```

32x24 位有符号部分 (1.31 x 1.23) [-1.0 .. 1.0) 到 34 位 (3.31) MAC, 四舍五入且无饱和。56 位 (2.54) 输出 $q0 \times p0$. [LH] 不对称地四舍五入到 $+\infty$, 截断并符号扩展为 34 位 (3.31)。然后将乘积覆盖、添加到 $q[49:16]$ 或从中减去 (即, q 中的 9.47 位定点值被截断为 3.31 位定点值)。最终的 34 位 (3.31) 结果是符号扩展的, 并用零填充到 56 位 (9.47) 定点值。

不太明白上面的R代表什么意思, 难道是四舍五入 (Round) 的意思?

C 语法:

C

```
1 ae_q56s AE_MULRFQ32SP24S_L (ae_q56s q0, ae_p24x2s p0);
2 void AE_MULARFQ32SP24S_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
3 void AE_MULSRFQ32SP24S_L (ae_q56s q /*inout*/, ae_q56s q0, ae_p24x2s p0);
```

2.8 加、减和比较操作

加、减和比较操作对应整数和小数应该都是相同的, 因为它是基于位操作的, **并且它的相加相减也不会像乘除法需要区分整数和小数操作。**

C

```
1 AE_ADDP24    p, p0, p1    [ slot1 ]
2 AE_SUBP24    p, p0, p1    [ slot1 ]
```

两个AE_PR寄存器 $p0$ 和 $p1$ 的逐元素加法/减法。结果放在 p 中。

两个寄存器的高低位都相加。

C 语法：

```
C
1  ae_p24x2s AE_ADDP24 (ae_p24x2s p0, ae_p24x2s p1);
2  ae_p24x2s AE_SUBP24 (ae_p24x2s p0, ae_p24x2s p1);
```

```
C
1  AE_ADDSP24S    p, p0, p1    [ slot1 ]
2  AE_SUBSP24S    p, p0, p1    [ slot1 ]
```

两个AE_PR寄存器 p0 和 p1 的元素级符号饱和加法/减法。结果放在 p 中。在饱和的情况下，状态 AE_OVERFLOW 设置为 1。

C 语法：

```
C
1  ae_p24x2s AE_ADDSP24S (ae_p24x2s p0, ae_p24x2s p1);
2  ae_p24x2s AE_SUBSP24S (ae_p24x2s p0, ae_p24x2s p1);
```

```
C
1  AE_NEGP24    p, p0          [ slot1 ]
```

AE_PR寄存器 p0 的元素级取反，结果放在 p 中。

C 语法：

```
C
1  ae_p24x2s AE_NEGP24 (ae_p24x2s p0);
```

C

```
1 AE_NEGSP24S    p, p0    [ slot1 ]
```

AE_PR寄存器 p0 的元素饱和否定，结果放在 p 中。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C 语法：

C

```
1 ae_p24x2s AE_NEGSP24S (ae_p24x2s p0);
```

C

```
1 AE_ABSP24    p, p0    [ slot1 ]
```

AE_PR寄存器的元素**绝对值** p0，结果以 p 为单位。

相当于绝对值函数。

C 语法：

C

```
1 ae_p24x2s AE_ABSP24 (ae_p24x2s p0);
```

C

```
1 AE_ABSSP24S    p, p0    [ slot1 ]
```

AE_PR寄存器的逐元素饱和绝对值，结果以 p 为单位。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C 语法：

C

```
1 ae_p24x2s AE_ABSSP24S (ae_p24x2s p0);
```

C

```
1 AE_MAXP24S  p, p0, p1  [ slot1 ]
2 AE_MINP24S  p, p0, p1  [ slot1 ]
```

两个AE_PR寄存器 p0 和 p1元素符号最大值/最小值。结果放在 p 中。

C语法:

C

```
1 ae_p24x2s AE_MAXP24S (ae_p24x2s p0, ae_p24x2s p1);
2 ae_p24x2s AE_MINP24S (ae_p24x2s p0, ae_p24x2s p1);
```

C

```
1 AE_MAXBP24S      p, p0, p1, bhl      [ slot1 ]
2 AE_MINBP24S      p, p0, p1, bhl      [ slot1 ]
```

计算两个AE_PR寄存器 (p0 和 p1) 的元素有符号最大值/最小值 p, 并清除或设置布尔寄存器对 bhl。**当且仅当 p0 的低 (L) 元素大于 (最大值AE_MAXBP24S函数) 或小于 (最小值AE_MINBP24S) p1 的低元素时, 才会设置 bhl[0]。**类似地, 当且仅当第一个输入AE_PR寄存器的高元素大于/小于寄存器的第二个输入AE_PR的高元素时, 才会设置 bhl[1]。

bhl有两个值, 表示H和L的标志位。AE_MAXBP24S用来识别最大值, 当最大值来自的寄存器 (p0

或p1) 变换的时候, bhl会发生变化, 同理, AE_MINBP24S 用来识别最小值, 当最小值来自的寄存器变换的时候, bhl会发生变化。

C语法:

C

```
1 void AE_MAXBP24S (ae_p24x2s p /*out*/, ae_p24x2s p0, ae_p24x2s p1, xtbool2 bhl
/*out*/);
2 void AE_MINBP24S (ae_p24x2s p /*out*/, ae_p24x2s p0, ae_p24x2s p1, xtbool2 bhl
/*out*/);
```

C

```
1 AE_MAXABSSP24S      p, p0, p1    [ slot1 ] (EP only)
2 AE_MINABSSP24S      p, p0, p1    [ slot1 ] (EP only)
```

两个AE_PR寄存器p0和p1的绝对值的元素最大值/最小值。结果放在 p 中。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法:

C

```
1 ae_p24x2s AE_MAXABSSP24S (ae_p24x2s p0, ae_p24x2s p1);
2 ae_p24x2s AE_MINABSSP24S (ae_p24x2s p0, ae_p24x2s p1);
```

C

```
1 AE_MAXABSSQ56S      q, q0, q1    [ slot1 ] (EP only)
2 AE_MINABSSQ56S      q, q0, q1    [ slot1 ] (EP only)
```

两个AE_QR寄存器q0和q1的绝对值的最大值/最小值，结果以q为单位。在饱和的情况下，状态AE_OVERFLOW设置为 1。

和上面的区别于是在AE_QR寄存器上面

C语法:

C

```
1 ae_q56s AE_MAXABSSQ56S (ae_q56s q0, ae_q56s q1);
2 ae_q56s AE_MINABSSQ56S (ae_q56s q0, ae_q56s q1);
```

C

```
1 AE_LTP24S          bhl, p0, p1    [ slot1 ]
```

在两个AE_PR寄存器p0和p1上进行元素**有符号小于比较**; 结果转到一对 BHL 相邻布尔寄存器。

从这里开始就进行比较功能了，这个函数比较的是p0是否小于p1，真的话bhl返回的值和假的话返回的值会有区别。

C语法：

```
C
1  xtbool2 AE_LTP24S (ae_p24x2s p0, ae_p24x2s p1);
```

```
C
1  AE_LEP24S      bhl, p0, p1      [ slot1 ]
```

在两个AE_PR寄存器p0和p1上进行元素符号**小于或相等**的比较; 结果转到一对 BHL 相邻布尔寄存器。

小于等于判断功能

C语法：

```
C
1  xtbool2 AE_LEP24S (ae_p24x2s p0, ae_p24x2s p1);
```

```
C
1  AE_EQP24      bhl, p0, p1      [ slot1 ]
```

两个AE_PR寄存器 p0 和 p1 上的元素相等比较; 结果转到一对 BHL 相邻布尔寄存器。

C语法：

```
C
1  xtbool2 AE_EQP24 (ae_p24x2s p0, ae_p24x2s p1);
```

C

```
1 AE_ADDQ56    q, q0, q1    [ slot1 ]
2 AE_SUBQ56    q, q0, q1    [ slot1 ]
```

两个AE_QR寄存器q0和q1加/减运算，结果放在q中。

C语法：

C

```
1 ae_q56s AE_ADDQ56 (ae_q56s q0, ae_q56s q1);
2 ae_q56s AE_SUBQ56 (ae_q56s q0, ae_q56s q1);
```

C

```
1 AE_ADDSQ56S   q, q0, q1    [ slot1 ]
2 AE_SUBSQ56S   q, q0, q1    [ slot1 ]
```

两个AE_QR寄存器q0和q1的符号饱和加法/减法，结果放在q中。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_q56s AE_ADDSQ56S (ae_q56s q0, ae_q56s q1);
2 ae_q56s AE_SUBSQ56S (ae_q56s q0, ae_q56s q1);
```

C

```
1 AE_NEGQ56     q, q0        [ slot1 ]
```

AE_QR寄存器的取反，结果放在q中。

C语法：

C

```
1 ae_q56s AE_NEGQ56 (ae_q56s q0);
```

```
C
```

```
1 AE_NEGSQ56S    q, q0      [ slot1 ]
```

寄存器q0 AE_QR饱和取反，结果放在q中。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

```
C
```

```
1 ae_q56s AE_NEGSQ56S (ae_q56s q0);
```

```
C
```

```
1 AE_ABSQ56    q, q0      [ slot1 ]
```

AE_QR寄存器的绝对值 q0，结果以 q 为单位。

C语法：

```
C
```

```
1 ae_q56s AE_ABSQ56 (ae_q56s q0);
```

```
C
```

```
1 AE_ABSSQ56S    q, q0      [ slot1 ]
```

AE_QR寄存器q0的饱和绝对值，结果以q为单位。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_q56s AE_ABSSQ56S (ae_q56s q0);
```

C

```
1 AE_MAXQ56S q, q0, q1 [ slot1 ]  
2 AE_MINQ56S q, q0, q1 [ slot1 ]
```

两个AE_QR寄存器 q0 和 q1 的带符号最大值/最小值，结果放在 q 中。

C语法：

C

```
1 ae_q56s AE_MAXQ56S (ae_q56s q0, ae_q56s q1);  
2 ae_q56s AE_MINQ56S (ae_q56s q0, ae_q56s q1);
```

C

```
1 AE_MAXBQ56S q, q0, q1, b [ slot1 ]  
2 AE_MINBQ56S q, q0, q1, b [ slot1 ]
```

计算两个AE_QR寄存器 q0 和 q1 的有符号最大值/最小 q，并清除或设置布尔寄存器 b。当且仅当 q0 大于（最大值）或小于（最小值）q1 时，才设置寄存器 b。

对于AE_MAXBQ56S，当q0大于q1的时候，才会设置b，对于AE_MINBQ56S同理。

C语法：

C

```
1 void AE_MAXBQ56S (ae_q56s q /*out*/, ae_q56s q0, ae_q56s q1, xtbool b /*out*/);  
2 void AE_MINBQ56S (ae_q56s q /*out*/, ae_q56s q0, ae_q56s q1, xtbool b /*out*/);
```

C

```
1 AE_LTQ56S    b, q0, q1    [ slot1 ]
```

在两个AE_QR寄存器q0和q1上带符号小于比较; 结果转到布尔寄存器 b。

C语法:

C

```
1 xtbool AE_LTQ56S (ae_q56s q0, ae_q56s q1);
```

C

```
1 AE_LEQ56S    b, q0, q1    [ slot1 ]
```

在两个AE_QR寄存器 q0 和 q1 上带符号小于或相等比较; 结果转到布尔寄存器 b。

C语法:

C

```
1 xtbool AE_LEQ56S (ae_q56s q0, ae_q56s q1);
```

C

```
1 AE_EQQ56     b, q0, q1    [ slot1 ]
```

两个AE_QR寄存器q0和q1上的相等比较; 结果转到布尔寄存器 b。

C语法:

C

```
1 xtbool AE_EQQ56 (ae_q56s q0, ae_q56s q1);
```

2.9 移位操作

HiFi 支持从即时、AR 寄存器文件或专用AE_SAR状态获取的移位量。使用 AR 寄存器或AE_SAR状态的移位指令将根据要移位的数据大小截断移位量。例如，将AE_PR寄存器移位 64 会将移位量从 64 截断为 0。

上面一句话的意思是移位的数据大小会决定移位量，比如上面的AE_PR寄存器就48bit，但是要移位64位，所以肯定是无法实现的，因此移位量会重置为0。

```
C
1 AE_SLLIP24 p, p0, i    [ slot1 ]
```

立即对AE_PR寄存器 p0 进行双路左移，结果放在 p 中：

- $p.L = p0.L \ll i$;
- $p.H = p0.H \ll i$.

C语法：

```
C
1 ae_p24x2s AE_SLLIP24 (ae_p24x2s p0, immediate i);
```

上面的参数有一个值得注意的，immediate，表示即时值。

```
C
1 AE_SRLIP24 p, p0, i    [ slot1 ]
```

立即对AE_PR寄存器p0进行双路逻辑（零扩展）右移，结果放在p中。

$p.L = p0.L \gg_{\text{u}} i$;

$p.H = p0.H \gg_{\text{u}} i$.

上面的U表示unsigned。

C语法：

```
C
1 ae_p24x2s AE_SRLIP24 (ae_p24x2s p0, immediate i);
```

C

```
1 AE_SRAIP24 p, p0, i [ slot1 ]
```

立即对AE_PR寄存器p0进行双路算术（符号扩展）右移，结果放在p中。

$p.L = p0.L \gg_s i;$

$p.H = p0.H \gg_s i.$

上面的S表示signed。

C语法：

C

```
1 ae_p24x2s AE_SRAIP24 (ae_p24x2s p0, immediate i);
```

C

```
1 AE_SLLISP24S p, p0, i [ slot1 ]
```

使用立即值对双路寄存器AE_PR p0的有符号饱和左移，结果放在p中。在饱和的情况下，状态AE_OVERFLOW 设置为 1。

C语法：

C

```
1 ae_p24x2s AE_SLLISP24S (ae_p24x2s p0, immediate i);
```

C

```
1 AE_SLLSP24 p, p0 [ slot1 ]
```

通过移位量寄存器AE_SAR对AE_PR寄存器p0进行双路左移，结果放在p中。

C语法：

```
C
1  ae_p24x2s AE_SLLSP24 (ae_p24x2s p0);
```

```
C
1  AE_SRLSP24  p, p0      [ slot1 ]
```

通过移位量寄存器AE_SAR对AE_PR寄存器p0进行双路逻辑（零扩展）右移，结果放在p中。

C语法：

```
C
1  ae_p24x2s AE_SRLSP24 (ae_p24x2s p0);
```

```
C
1  AE_SRASP24  p, p0      [ slot1 ]
```

通过移位量寄存器AE_SAR对AE_PR寄存器p0进行双向**算术（符号扩展）右移**，结果放在p中。

这个函数和之前的区别是[知算术移位](#)。

C语法：

```
C
1  ae_p24x2s AE_SRASP24 (ae_p24x2s p0);
```

```
C
1  AE_SLLSSP24S    p, p0      [ slot1 ]
```

通过移位量寄存器AE_SAR对AE_PR寄存器p0进行双向有符号饱和左移，结果放在p中。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

```
C
1  ae_p24x2s AE_SLLSP24S (ae_p24x2s p0);
```

```
C
1  AE_SLLIQ56  q, q0, i    [ slot0, Inst ]
```

使用立即值对AE_QR寄存器q0的左移，结果放在q中。

C语法：

```
C
1  ae_q56s AE_SLLIQ56 (ae_q56s q0, immediate i);
```

```
C
1  AE_SRLIQ56  q, q0, i    [ slot0, Inst ]
```

使用立即值对AE_QR寄存器q0的逻辑（零扩展）右移，结果放在q中。

C语法：

```
C
1  ae_q56s AE_SRLIQ56 (ae_q56s q0, immediate i);
```

C

```
1 AE_SRAIQ56 q, q0, i [ slot0, Inst ]
```

使用立即值对AE_QR寄存器q0的算术（符号扩展）右移，结果放在q中。

C语法：

C

```
1 ae_q56s AE_SRAIQ56 (ae_q56s q0, immediate i);
```

C

```
1 AE_SLLISQ56S q, q0, i [ slot0, Inst ]
```

使用立即值对AE_QR寄存器q0进行带符号饱和左移，结果放在q中。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_q56s AE_SLLISQ56S (ae_q56s q0, immediate i);
```

C

```
1 AE_SLLAQ56 q, q0, a [ slot0, Inst ]
```

通过AR寄存器a对AE_QR寄存器q0进行左移，结果放在q中。

C语法：

C

```
1 ae_q56s AE_SLLAQ56 (ae_q56s q0, unsigned a);
```

C

```
1 AE_SRLAQ56 q, q0, a [ slot0, Inst ]
```

通过AR寄存器a对AE_QR寄存器q0进行逻辑（零扩展）右移，结果放在q中。

C语法：

C

```
1 ae_q56s AE_SRLAQ56 (ae_q56s q0, unsigned a);
```

C

```
1 AE_SRAAQ56 q, q0, a [ slot0, Inst ]
```

通过AR寄存器a对AE_QR寄存器q0进行算术（符号扩展）右移，结果放在q中。

C语法：

C

```
1 ae_q56s AE_SRAAQ56 (ae_q56s q0, unsigned a);
```

C

```
1 AE_SLLASQ56S q, q0, a [ slot0, Inst ]
```

通过AR寄存器a对寄存器q0进行AE_QR符号饱和左移，结果放在q中。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_q56s AE_SLLAQ56S (ae_q56s q0, unsigned a);
```

C

```
1 AE_SLLSQ56 q, q0 [ slot0, Inst ]
```

通过移位量寄存器 AE_SAR 对 AE_QR 寄存器q0的左移，结果放在q中。

C语法：

C

```
1 ae_q56s AE_SLLSQ56 (ae_q56s q0);
```

C

```
1 AE_SRLSQ56 q, q0 [ slot0, Inst ]
```

通过移位量寄存器AE_SAR对AE_QR寄存器q0进行逻辑右移，结果放在q中。

C语法：

C

```
1 ae_q56s AE_SRLSQ56 (ae_q56s q0);
```

C

```
1 AE_SRASQ56 q, q0 [ slot0, Inst ]
```

通过移位量寄存器AE_SAR对AE_QR寄存器q0进行算术右移，结果放在q中。

C语法：

C

```
1 ae_q56s AE_SRASQ56 (ae_q56s q0);
```

C

```
1 AE_SLLSQ56S q, q0 [ slot0, Inst ]
```

通过移位量寄存器AE_SAR对AE_QR寄存器q0进行带符号饱和左移，结果放在q中。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_q56s AE_SLLSQ56S (ae_q56s q0);
```

C

```
1 AE_SLAASQ56S q, q0, a [ slot0 ] (EP only)
```

通过AR寄存器a对AE_QR寄存器q0进行双路饱和算术移位，结果放在q中。该操作使用寄存器 a 的最高有效位来确定移位量的符号，并使用六个最低有效位作为实际移位量，从而产生移位量范围为 -63 到 +63。如果左移时出现 56 位饱和，则状态AE_OVERFLOW设置为 1。

$q = q0 \ll a$, if $a \geq 0$;

$q = q0 \gg_s(-a)$, if $a < 0$

C语法：

C

```
1 ae_q56s AE_SLAASQ56S (ae_q56s q0, int a);
```

2.10 规范化移位量操作

C

```
1 AE_NSAQ56S a, q0 [ slot0, Inst ]
```

计算左移量，该量将归一化AE_QR寄存器的两者补码内容，并将量（在 0 到 55 范围内）写入 AR 寄存器 a。如果 q0 包含 0 或 -1，则返回 55。要计算 9.47 定点数的归一化指数，请从结果中减去 8。如果结果为负数，则需要右移才能进行归一化。

C语法：

C

```
1 int AE_NSAQ56S (ae_q56s q0);
```

2.11 截断、四舍五入（round）、饱和、转换和移动操作

C

```
1 AE_TRUNCA32Q48 a, q0 [ slot0, Inst ]
```

将 48 位（1.47）AE_QR分数 q0 截断为 32 位（1.31）AR 分数 a。

C语法：

C

```
1 int AE_TRUNCA32Q48 (ae_q56s q0);
```

C

```
1 AE_TRUNCP24Q48X2 p, qh, ql [ slot1 ]
```

将寄存器AE_QR qh 和 ql 中的两个 48 位分数（1.47）截断为寄存器AE_PR p 中的两个 24 位分数（1.23）元素。

注意：C 固有AE_TRUNCP24Q48截断并复制AE_QR中的单个 48 位分数到AE_PR中的两个 24 位分数。它是通过AE_TRUNCP24Q48X2操作实现的。

使用两个AE_QR寄存器（累加器）组成类似AE_PR的组合，然后再截断。

C语法：

```
C
1 ae_p24x2s AE_TRUNC24Q48X2 (ae_q56s qh, ae_q56s ql);
2 ae_p24x2s AE_TRUNC24Q48 (ae_q56s q0);
```

```
C
1 AE_TRUNC24A32X2    p, ah, al    [ slot0, Inst ]
```

将 AR 寄存器 ah 和 al 中的两个 32 位分数（1.31）截断为寄存器AE_PR p 中的两个 24 位分数（1.23）元素。

C语法：

```
C
1 ae_p24x2s AE_TRUNC24A32X2 (unsigned ah, unsigned al);
```

```
C
1 AE_TRUNC16P24S.L (.H) a, p0    [ slot0, Inst ]
```

截断并符号扩展 24 位（1.23）AE_PR分数中的p0.L（p0.H）到 AR 中的 16 位定点数（1.15）。

这个原文里面有一些错误，原文：Truncate and sign-extend a 24-bit (1.23) AE_PR fraction in p0.L (p0.H) to a 16-bit fixed point number (17.15) in AR. 里面的16位定点值表示17.15是错误的，应该去掉7。

这里还有一个有趣的地方，就是24位和16位的分数英文表示是不一样的，24位的是“fraction”，16位的有“fixed point”前缀，但两个其实都是表述分数，这个说法我在下面的函数里面也看到了类似的情况。

其实都是在表述定点数，只是叫法不同而已。

C语法：

C

```
1 int AE_TRUNCA16P24S_L (ae_p24x2s p0);
```

C

```
1 AE_TRUNCP16          p, p0          [ slot1 ]
```

将 p0 中的两个 24 位分数（1.23）截断为 p 中的 16 位定点分数（1.23，8 个最低有效位设置为零）。

看来16位的数值才能被称为定点，24位是不算的？

定点只是一种称呼，和多少位没有关系，无论是16位，24位，还是32位，都有定点数。定点数本质是用整数表示分数（小数）的一种方法。

C语法：

C

```
1 ae_p24x2s AE_TRUNCP16 (ae_p24x2s p0);
```

C

```
1 AE_TRUNCQ32          q, q0          [ slot1 ]
```

截断 56 位定点数（9.47）q0 的 **16 个最低有效位**，并将结果放在 q 中。

C语法：

C

```
1 ae_q56s AE_TRUNCQ32 (ae_q56s q0);
```

C

```
1 AE_ROUNDSP16SYM      p, p0      [ slot1 ]
```

将每个 p0 元素的最低有效 8 位四舍五入，并将值饱和为 p 中的 16 位分数 [-1.0 .. 1.0) 。**舍入是对称的，两半从零四舍五入，即最低显著结果位的 0.5 倍向上舍入到 1.0（并饱和）， -0.5 倍最不显著的结果位向下舍入到 -1.0。**在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_p24x2s AE_ROUNDSP16SYM (ae_p24x2s p0);
```

C

```
1 AE_ROUNDSP16ASYM      p, p0      [ slot1 ]
```

将每个 p0 元素的最低有效 8 位四舍五入，并将值饱和为 p 中的 16 位分数 [-1.0 .. 1.0) 。**舍入是不对称的，一半向上舍入，即最低有效结果位的 0.5 倍向上舍入到 1.0（并饱和）， -0.5 倍的最低有效结果位四舍五入为 0。**在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_p24x2s AE_ROUNDSP16ASYM (ae_p24x2s p0);
```

C

```
1 AE_ROUNDSP24Q48SYM    p, q0      [ slot1 ]
```

将 q0 的最低有效 24 位四舍五入，并将该值饱和为 24 位分数 [-1.0 .. 1.0) ，将结果复制两次放在 p 中（分别放入H和L元素）。舍入是对称的，两半从零四舍五入，即最低显著结果位的 0.5 倍向上舍入到 1.0（并饱和）， -0.5 倍最不显著的结果位向下舍入到 -1.0。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_p24x2s AE_ROUNDSP24Q48SYM (ae_q56s q0);
```

C

```
1 AE_ROUNDSP24Q48ASYM p, q0 [ slot1 ]
```

将 q0 的最低有效 24 位四舍五入，并将该值饱和为 24 位分数 [-1.0 .. 1.0)，将结果复制两次放在 p 中。舍入是不对称的，一半向上舍入，即最低有效结果位的 0.5 倍向上舍入到 1.0（并饱和），-0.5 倍的最低有效结果位四舍五入为 0。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_p24x2s AE_ROUNDSP24Q48ASYM (ae_q56s q0);
```

C

```
1 AE_ROUNDSP16Q48SYM p, q0 [ slot1 ]
```

将 q0 的最低有效 32 位四舍五入，并将该值饱和为 16 位分数 [-1.0 .. 1.0)，将结果复制两次放在 p 中。舍入是对称的，两半从零四舍五入，即最低显著结果位的 0.5 倍向上舍入到 1.0（并饱和），-0.5 倍最不显著的结果位向下舍入到 -1.0。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_p24x2s AE_ROUNDSP16Q48SYM (ae_q56s q0);
```

C

```
1 AE_ROUNDSP16Q48ASYM p, q0 [ slot1 ]
```

将 q0 的最低有效 32 位四舍五入，并将该值饱和为 16 位分数 [-1.0 .. 1.0)，将结果复制两次放在 p 中。舍入是不对称的，一半向上舍入，即最低有效结果位的 0.5 倍向上舍入到 1.0（并饱和），-0.5 倍的最低有效结果位四舍五入为 0。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

```
C
1  ae_p24x2s AE_ROUNDSP16Q48ASYM (ae_q56s q0);
```

```
C
1  AE_ROUNDQ32SYM      q, q0      [ slot1 ]
```

将 q0 的最低有效 16 位四舍五入，并将该值饱和为 32 位分数 [-1.0 .. 1.0)，将结果放在 q 中。舍入是对称的，两半从零四舍五入，即最低显著结果位的 0.5 倍向上舍入到 1.0（并饱和），-0.5 倍最不显著的结果位向下舍入到 -1.0。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

```
C
1  ae_q56s AE_ROUNDQ32SYM (ae_q56s q0);
```

```
C
1  AE_ROUNDQ32ASYM      q, q0      [ slot1 ]
```

将 q0 的最低有效 16 位四舍五入，并将该值饱和为 32 位分数 [-1.0 .. 1.0)，将结果放在 q 中。舍入是不对称的，一半向上舍入，即最低有效结果位的 0.5 倍向上舍入到 1.0（并饱和），-0.5 倍的最低有效结果位四舍五入为 0。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

```
C
1  ae_q56s AE_ROUNDQ32ASYM (ae_q56s q0);
```


C

```
1 AE_SATQ48S      q, q0      [ slot1 ]
```

将 q0 中的值饱和为 48 位 (1.47) 分数，并将结果置于 q 中。在饱和的情况下，状态AE_OVERFLOW设置为 1。

C语法：

C

```
1 ae_q56s AE_SATQ48S (ae_q56s q0);
```

C

```
1 AE_CVTP24A16X2.LL (.LH, .HL. HH)    p, ah, al    [ slot0, Inst ]
```

复制 16 个most (.HL, .HH) 或least (.LL, .LH) 有效位从AR寄存器ah变成p.H的最有效位，以及16个most (.LH, .HH) 或least (.LL, .HL) 有效位从AR寄存器al转换为p.L的最有效位。

这个操作有点不理解，原文“Copy the 16 most (.HL, .HH) or least (.LL, .LH) significant bits from the AR register ah into the most significant bits of p.H, and the 16 most (.LH, .HH) or least (.LL, .HL) significant bits from the AR register al into the most significant bits of p.L.”

注意：C 固有AE_CVTP24A16X2等效于并通过操作 AE_CVTP24A16X2.LL 。C 固有AE_CVTP24A16将 AR 寄存器中的 16 个最低有效位复制到AE_PR寄存器两个元素的 16 个最高有效位中。它是通过操作 AE_CVTP24A16X2.LL 实现的。

相当于在此基础上有一些进化，AE_CVTP24A16X2和AE_CVTP24A16都是AE_CVTP24A16X2_LL相关函数的一些简化操作，前者相当是AE_CVTP24A16X2_LL，后者相当于AE_CVTP24A16X2_LL的一些操作。

C语法：

```
C
1 ae_p24x2s AE_CVTP24A16X2_LL (unsigned ah, unsigned al);
2 ae_p24x2s AE_CVTP24A16X2 (unsigned ah, unsigned al);
3 ae_p24x2s AE_CVTP24A16 (unsigned a);
```

```
C
1 AE_CVTQ48A32S      q, a      [ slot0, Inst ]
```

将 AR 寄存器 a 中的 32 位分数 (1.31) 转换为 q 中的 56 位AE_QR定点数 (9.47) 。

分数和定点数在这里是一个东西，不需要太困惑

C语法：

```
C
1 ae_q56s AE_CVTQ48A32S (unsigned a);
```

```
C
1 AE_CVTQ48P24S.L (.H)    q, p0      [ slot1 ]
```

在 p0.L (p0.H) 中转换 24 位 (1.23) 值到 q 中的 48 位 (1.47) 有符号分数。

C语法：

```
C
1 ae_q56s AE_CVTQ48P24S_L (ae_p24x2s p0);
```

```
C
1 AE_CVTA32P24.L (.H)    a, p0      [ slot0, Inst ]
```

在 p0.L (p0.H) 中转换 24 位 (1.23) 值到 AR 寄存器 a 中的 32 位 (1.31) 有符号分数。

C语法:

```
C
1  int AE_CVTA32P24_L (ae_p24x2s p0);
```

```
C
1  AE_MOVPA24X2      p, ah, al  [ slot0, Inst ]
```

将两个 AR 寄存器中每个寄存器中的 24 个最低有效位 ah 和 al 复制到AE_PR寄存器 p 的两个元素中。

注意: C 固有AE_MOVPA24将 AR 寄存器的 24 个最低有效位复制到AE_PR寄存器的两个元素中。它是通过AE_MOVPA24X2行动实施的。

相当于放两个相同的值到AE_PR的两个元素中。

C语法:

```
C
1  ae_p24x2s AE_MOVPA24X2 (unsigned ah, unsigned al);
2  ae_p24x2s AE_MOVPA24 (unsigned a);
```

```
C
1  AE_MOVAP24S.L (.H) a, p0      [ slot0, Inst ]
```

从 p0.L (p0.H) 复制并符号扩展 24 位值 到 AR 寄存器 a。

C语法:

```
C
1  int AE_MOVAP24S_L (ae_p24x2s p0);
```

```
C
1 AE_MOVP48      p, p0      [ slot0, slot1, Inst ]
```

将 p0 的内容复制到 p。

C语法：

```
C
1 ae_p24x2s AE_MOVP48 (ae_p24x2s p0);
```

```
C
1 AE_MOVQ56      q, q0      [ slot0, slot1, Inst ]
```

将 q0 的内容复制到 q。

C语法：

```
C
1 ae_q56s AE_MOVQ56 (ae_q56s q0);
```

```
C
1 AE_MOVTP48      p, p0, b    [ slot1 ]
```

如果设置了 b，则将 p0 的内容复制到 p。

C语法：

```
C
1 void AE_MOVTP48 (ae_p24x2s p /*inout*/, ae_p24x2s p0, xtbool b);
```

C

```
1 AE_MOVFP48      p, p0, b    [ slot1 ]
```

如果 b 是没有设置的（clear），则将 p0 的内容复制到 p。

C语法：

C

```
1 void AE_MOVFP48 (ae_p24x2s p /*inout*/, ae_p24x2s p0, xtbool b);
```

C

```
1 AE_MOVTP24X2      p, p0, bhl [ slot1 ]
```

如果设置了 bhl[0]，则复制 p0.L 的内容至 p.L;

如果设置了 bhl[1]，请复制 P0.H 的内容到 P.H.

C语法：

C

```
1 void AE_MOVTP24X2 (ae_p24x2s p /*inout*/, ae_p24x2s p0, xtbool2 bhl);
```

C

```
1 AE_MOVFP24X2      p, p0, bhl [ slot1 ]
```

如果 bhl[0] 是没有被设置的（clear），则复制 p0.L 的内容至 p.L;

如果 bhl[1] 是没有被设置的（clear），则复制 p0.H 的内容至 p.H;

C语法：

C

```
1 void AE_MOVFP24X2 (ae_p24x2s p /*inout*/, ae_p24x2s p0, xtbool2 bhl);
```

C

```
1 AE_MOVTQ56      q, q0, b    [ slot0, Inst ]
```

如果设置了 b，则将 q0 的内容复制到 q。

C语法：

C

```
1 void AE_MOVTQ56 (ae_q56s q /*inout*/, ae_q56s q0, xtbool b);
```

C

```
1 AE_MOVFQ56      q, q0, b    [ slot0, Inst ]
```

如果 b 是没有被设置的（clear），则将 q0 的内容复制到 q。

C语法：

C

```
1 void AE_MOVFQ56 (ae_q56s q /*inout*/, ae_q56s q0, xtbool b);
```

2.12 选择和排列操作

C

```
1 AE_SELP24.LL p, p0, p1 [ slot1 ][ slot0 (EP only) ]
```

p.H = p0.L;

p.L = p1.L.

C语法:

```
C
1 ae_p24x2s AE_SEL24_LL (ae_p24x2s p0, ae_p24x2s p1);
```

```
C
1 AE_SEL24.LH p, p0, p1 [ slot1 ][ slot0 (EP only) ]
```

p.H = p0.L;

p.L = p1.H.

C语法:

```
C
1 ae_p24x2s AE_SEL24_LH (ae_p24x2s p0, ae_p24x2s p1);
```

```
C
1 AE_SEL24.HL p, p0, p1 [ slot1 ][ slot0 (EP only) ]
```

p.H = p0.H;

p.L = p1.L.

C语法:

```
C
1 ae_p24x2s AE_SEL24_HL (ae_p24x2s p0, ae_p24x2s p1);
```

C

```
1 AE_SELP24.HH p, p0, p1 [ slot1 ][ slot0 (EP only) ]
```

p.H = p0.H;

p.L = p1.H.

C语法:

C

```
1 ae_p24x2s AE_SELP24_HH (ae_p24x2s p0, ae_p24x2s p1);
```

2.13 按位逻辑操作

这些操作执行的计算由其操作码助记符和操作数暗示，如下所示。

C

```
1 AE_ANDP48    p, p0, p1    [ slot1 ]
2 AE_NANDP48   p, p0, p1    [ slot1 ]
3 AE_ORP48     p, p0, p1    [ slot1 ]
4 AE_XORP48    p, p0, p1    [ slot1 ]
5 AE_ANDQ56    q, q0, q1    [ slot1 ]
6 AE_NANDQ56   q, q0, q1    [ slot1 ]
7 AE_ORQ56     q, q0, q1    [ slot1 ]
8 AE_XORQ56    q, q0, q1    [ slot1 ]
```

注意：C 内部函数AE_NOTP48和AE_NOTQ56分别通过操作AE_NANDP48和AE_NANDQ56实现。

C语法:

C

```
1 ae_p24x2s AE_ANDP48 (ae_p24x2s p0, ae_p24x2s p1);
2 ae_p24x2s AE_NANDP48 (ae_p24x2s p0, ae_p24x2s p1);
3 ae_p24x2s AE_ORP48 (ae_p24x2s p0, ae_p24x2s p1);
4 ae_p24x2s AE_XORP48 (ae_p24x2s p0, ae_p24x2s p1);
5 ae_p24x2s AE_NOTP48 (ae_p24x2s p0);
6
7 ae_q56s AE_ANDQ56 (ae_q56s q0, ae_q56s q1);
8 ae_q56s AE_NANDQ56 (ae_q56s q0, ae_q56s q1);
9 ae_q56s AE_ORQ56 (ae_q56s q0, ae_q56s q1);
```



```

10 ae_q56s AE_XORQ56 (ae_q56s q0, ae_q56s q1);
11 ae_q56s AE_NOTQ56 (ae_q56s q0);

```

2.14 位反转 (EP only)

C

```

1 AE_ADDBRBA32    a, ab, ax    [ slot0 ]

```

32 位添加到位反转基数:

$a \leftarrow \text{bitrev}_{32}(\text{bitrev}_{32}(ab) + ax)$.

此帮助程序操作可与索引加载和存储 (.X) 在优化的 FFT 实现中执行位反向寻址。例如，下面的 C 代码通过 256 个 16 位复杂数据元素块以位反转顺序进行访问：

C

```

1  /* The data elements will be accessed in the following order:
2  0, 128, 64, 192, 32, 160, 96, 224, 16, 144, 80, ...
3  i.e., for i = 0...255, access element at index bitrev8(i). */
4  ae_p16x2s *buf = ...;
5  unsigned int index = 0;
6  unsigned int stride =
7      0x80000000U >> (8 /* log2256 */ + 2 /* 4-byte alignment */ - 1);
8  ...
9  for (...) {
10     ...
11     ae_p24x2s p = AE_LP16X2F_X(buf, index);
12     index = AE_ADDBRBA32(index, stride);
13     ...
14 }

```

C语法:

C

```

1  unsigned AE_ADDBRBA32 (unsigned ab, unsigned ax);

```

2.15 置零操作

C

```
1 AE_ZEROP48 p [ slot1 ]
```

将AE_PR寄存器 p 的所有位设置为零。

C语法：

C

```
1 ae_p24x2s AE_ZEROP48 (void);
```

C

```
1 AE_ZEROQ56 q [ slot1 ]
```

将AE_QR寄存器 q 的所有位设置为零。

C语法：

C

```
1 ae_q56s AE_ZEROQ56 (void);
```

2.16 比特流和可变长度编码和解码指令

关于里面的可变长度编码其实就是霍夫曼编码，但是这一块我并不是很了解，后面了解了具体情况再进行该章节的更新。

2.17 指令列表 – 显示插槽分配

下面列出了可以在Inst, slot0和slot1中发出的所有HiFi 2 / EP指令的摘要。（[EP] 表示仅适用于 HiFi 2 EP）

后面等2.16完成会把下面的所有函数都标记上链接地址。

2.17.1 Inst

AE_CVTA32P24.H, AE_CVTA32P24.L, AE_CVTP24A16X2.HH, AE_CVTP24A16X2.HL, AE_CVTP24A16X2.LH, AE_CVTP24A16X2.LL, AE_CVTQ48A32S, AE_DB, AE_DBI, AE_LB, AE_LBI, AE_LBK, AE_LBKI, AE_LP16F.I, AE_LP16F.IU, AE_LP16F.X, AE_LP16F.XU, AE_LP16X2F.I, AE_LP16X2F.IU, AE_LP16X2F.X, AE_LP16X2F.XU, AE_LP24.I, AE_LP24.IU, AE_LP24.X, AE_LP24.XU, AE_LP24F.I, AE_LP24F.IU, AE_LP24F.X, AE_LP24F.XU, AE_LP24X2.I, AE_LP24X2.IU, AE_LP24X2.X, AE_LP24X2.XU, AE_LP24X2F.I, AE_LP24X2F.IU, AE_LP24X2F.X, AE_LP24X2F.XU, AE_LQ32F.I, AE_LQ32F.IU, AE_LQ32F.X, AE_LQ32F.XU, AE_LQ56.I, AE_LQ56.IU, AE_LQ56.X, AE_LQ56.XU, AE_MOVAP24S.H, AE_MOVAP24S.L, AE_MOVFQ56, AE_MOVP48, AE_MOVPA24X2, AE_MOVQ56, AE_MOVTQ56, AE_NSAQ56S, AE_SB, AE_SBF, AE_SBI, AE_SHA32, AE_SLLAQ56, AE_SLLASQ56S, AE_SLLIQ56, AE_SLLISQ56S, AE_SLLSQ56, AE_SLLSSQ56S, AE_SP16F.L.I, AE_SP16F.L.IU, AE_SP16F.L.X, AE_SP16F.L.XU, AE_SP16X2F.I, AE_SP16X2F.IU, AE_SP16X2F.X, AE_SP16X2F.XU, AE_SP24F.L.I, AE_SP24F.L.IU, AE_SP24F.L.X, AE_SP24F.L.XU, AE_SP24S.L.I, AE_SP24S.L.IU, AE_SP24S.L.X, AE_SP24S.L.XU, AE_SP24X2F.I, AE_SP24X2F.IU, AE_SP24X2F.X, AE_SP24X2F.XU, AE_SP24X2S.I, AE_SP24X2S.IU, AE_SP24X2S.X, AE_SP24X2S.XU, AE_SQ32F.I, AE_SQ32F.IU, AE_SQ32F.X, AE_SQ32F.XU, AE_SQ56S.I, AE_SQ56S.IU, AE_SQ56S.X, AE_SQ56S.XU, AE_SRAAQ56, AE_SRAIQ56, AE_SRASQ56, AE_SRLAQ56, AE_SRLIQ56, AE_SRLSQ56, AE_TRUNCA16P24S.H, AE_TRUNCA16P24S.L, AE_TRUNCA32Q48, AE_TRUNCP24A32X2, AE_VLDL16C, AE_VLDL16T, AE_VLDL32T, AE_VLDSHT, AE_VLEL16T, AE_VLEL32T, AE_VLES16C, RUR.AE_BITHEAD, RUR.AE_BITPTR, RUR.AE_BITSUSED, RUR.AE_CBEGIN0[EP], RUR.AE_CEND0[EP], RUR.AE_FIRST_TS, RUR.AE_NEXTOFFSET, RUR.AE_OVERFLOW, RUR.AE_OVF_SAR, RUR.AE_SAR, RUR.AE_SD_NO, RUR.AE_SEARCHDONE, RUR.AE_TABLESIZE, RUR.AE_TS_FTS_BU_BP, WUR.AE_BITHEAD, WUR.AE_BITPTR, WUR.AE_BITSUSED, WUR.AE_CBEGIN0[EP], WUR.AE_CEND0[EP], WUR.AE_FIRST_TS, WUR.AE_NEXTOFFSET, WUR.AE_OVERFLOW, WUR.AE_OVF_SAR, WUR.AE_SAR, WUR.AE_SD_NO, WUR.AE_SEARCHDONE, WUR.AE_TABLESIZE, WUR.AE_TS_FTS_BU_BP

2.17.2 slot0

AE_ADDBRBA32[EP], AE_CVTA32P24.H, AE_CVTA32P24.L, AE_CVTP24A16X2.HH, AE_CVTP24A16X2.HL, AE_CVTP24A16X2.LH, AE_CVTP24A16X2.LL, AE_CVTQ48A32S, AE_LP16F.C[EP], AE_LP16F.I, AE_LP16F.IU, AE_LP16F.X, AE_LP16F.XU, AE_LP16X2F.C[EP], AE_LP16X2F.I, AE_LP16X2F.IU, AE_LP16X2F.X, AE_LP16X2F.XU, AE_LP24.C[EP], AE_LP24.I, AE_LP24.IU, AE_LP24.X, AE_LP24.XU, AE_LP24F.C[EP], AE_LP24F.I, AE_LP24F.IU, AE_LP24F.X, AE_LP24F.XU, AE_LP24X2.C[EP], AE_LP24X2.I, AE_LP24X2.IU, AE_LP24X2.X, AE_LP24X2.XU, AE_LP24X2F.C[EP], AE_LP24X2F.I, AE_LP24X2F.IU, AE_LP24X2F.X, AE_LP24X2F.XU, AE_LQ32F.C[EP], AE_LQ32F.I, AE_LQ32F.IU, AE_LQ32F.X, AE_LQ32F.XU, AE_LQ56.C[EP], AE_LQ56.I, AE_LQ56.IU, AE_LQ56.X, AE_LQ56.XU, AE_MOVAP24S.H, AE_MOVAP24S.L, AE_MOVP48, AE_MOVPA24X2, AE_MOVQ56, AE_MOVTQ56, AE_NSAQ56S, AE_SELP24.HH[EP], AE_SELP24.HL[EP], AE_SELP24.LH[EP], AE_SELP24.LL[EP], AE_SLAASQ56S[EP], AE_SLLAQ56, AE_SLLASQ56S, AE_SLLIQ56, AE_SLLISQ56S, AE_SLLSQ56, AE_SLLSSQ56S, AE_SP16F.L.C[EP], AE_SP16F.L.I, AE_SP16F.L.IU, AE_SP16F.L.X, AE_SP16F.L.XU, AE_SP16X2F.C[EP], AE_SP16X2F.I, AE_SP16X2F.IU, AE_SP16X2F.X, AE_SP16X2F.XU, AE_SP24F.L.C[EP], AE_SP24F.L.I, AE_SP24F.L.IU, AE_SP24F.L.X, AE_SP24F.L.XU, AE_SP24S.L.C[EP], AE_SP24S.L.I, AE_SP24S.L.IU, AE_SP24S.L.X, AE_SP24S.L.XU, AE_SP24X2F.C[EP], AE_SP24X2F.I, AE_SP24X2F.IU, AE_SP24X2F.X, AE_SP24X2F.XU, AE_SP24X2S.C[EP], AE_SP24X2S.I, AE_SP24X2S.IU, AE_SP24X2S.X, AE_SP24X2S.XU, AE_SQ32F.C[EP], AE_SQ32F.I, AE_SQ32F.IU, AE_SQ32F.X, AE_SQ32F.XU, AE_SQ56S.C[EP], AE_SQ56S.I, AE_SQ56S.IU, AE_SQ56S.X, AE_SQ56S.XU, AE_SRAAQ56, AE_SRAIQ56, AE_SRASQ56, AE_SRLAQ56, AE_SRLIQ56, AE_SRLSQ56, AE_TRUNCA16P24S.H, AE_TRUNCA16P24S.L, AE_TRUNCA32Q48, AE_TRUNCP24A32X2

2.17.3 slot1

AE_ABSP24, AE_ABSQ56, AE_ABSSP24S, AE_ABSSQ56S, AE_ADDP24, AE_ADDQ56, AE_ADDSP24S,
AE_ADDSQ56S, AE_ANDP48, AE_ANDQ56, AE_CVTQ48P24S.H, AE_CVTQ48P24S.L, AE_EQP24, AE_EQQ56,
AE_LEP24S, AE_LEQ56S, AE_LTP24S, AE_LTQ56S, AE_MAXABSSP24S[EP], AE_MAXABSSQ56S[EP],
AE_MAXBP24S, AE_MAXBQ56S, AE_MAXP24S, AE_MAXQ56S, AE_MINABSSP24S[EP], AE_MINABSSQ56S[EP],
AE_MINBP24S, AE_MINBQ56S, AE_MINP24S, AE_MINQ56S, AE_MOVFP24X2, AE_MOVFP48, AE_MOVP48,
AE_MOVQ56, AE_MOVTP24X2, AE_MOVTP48, AE_MULAAFP24S.HH.LL, AE_MULAAFP24S.HL.LH,
AE_MULAAP24S.HH.LL, AE_MULAAP24S.HL.LH, AE_MULAFP24S.HH, AE_MULAFP24S.HL, AE_MULAFP24S.LH,
AE_MULAFP24S.LL, AE_MULAFQ32SP16S.H, AE_MULAFQ32SP16S.L, AE_MULAFQ32SP16U.H,
AE_MULAFQ32SP16U.L, AE_MULAFQ32SP24S.H [EP], AE_MULAFQ32SP24S.L [EP], AE_MULAFS32P16S.HH,
AE_MULAFS32P16S.HL, AE_MULAFS32P16S.LH, AE_MULAFS32P16S.LL, AE_MULAFS56P24S.HH,
AE_MULAFS56P24S.HL, AE_MULAFS56P24S.LH, AE_MULAFS56P24S.LL, AE_MULAP24S.HH,
AE_MULAP24S.HL, AE_MULAP24S.LH, AE_MULAP24S.LL, AE_MULAQ32SP16S.H, AE_MULAQ32SP16S.L,
AE_MULAQ32SP16U.H, AE_MULAQ32SP16U.L, AE_MULARFQ32SP24S.H [EP], AE_MULARFQ32SP24S.L [EP],
AE_MULAS56P24S.HH, AE_MULAS56P24S.HL, AE_MULAS56P24S.LH, AE_MULAS56P24S.LL,
AE_MULASFP24S.HH.LL, AE_MULASFP24S.HL.LH, AE_MULASP24S.HH.LL, AE_MULASP24S.HL.LH,
AE_MULFP24S.HH, AE_MULFP24S.HL, AE_MULFP24S.LH, AE_MULFP24S.LL, AE_MULFQ32SP16S.H,
AE_MULFQ32SP16S.L, AE_MULFQ32SP16U.H, AE_MULFQ32SP16U.L, AE_MULFQ32SP24S.H [EP],
AE_MULFQ32SP24S.L [EP], AE_MULFS32P16S.HH, AE_MULFS32P16S.HL, AE_MULFS32P16S.LH,
AE_MULFS32P16S.LL, AE_MULP24S.HH, AE_MULP24S.HL, AE_MULP24S.LH, AE_MULP24S.LL,
AE_MULQ32SP16S.H, AE_MULQ32SP16S.L, AE_MULQ32SP16U.H, AE_MULQ32SP16U.L,
AE_MULRFQ32SP24S.H [EP], AE_MULRFQ32SP24S.L [EP], AE_MULSAFP24S.HH.LL, AE_MULSAFP24S.HL.LH,
AE_MULSAP24S.HH.LL, AE_MULSAP24S.HL.LH, AE_MULSFP24S.HH, AE_MULSFP24S.HL, AE_MULSFP24S.LH,
AE_MULSFP24S.LL, AE_MULSFQ32SP16S.H, AE_MULSFQ32SP16S.L, AE_MULSFQ32SP16U.H,
AE_MULSFQ32SP16U.L, AE_MULSFQ32SP24S.H [EP], AE_MULSFQ32SP24S.L [EP], AE_MULSFS32P16S.HH,
AE_MULSFS32P16S.HL, AE_MULSFS32P16S.LH, AE_MULSFS32P16S.LL, AE_MULSFS56P24S.HH,
AE_MULSFS56P24S.HL, AE_MULSFS56P24S.LH, AE_MULSFS56P24S.LL, AE_MULSP24S.HH, AE_MULSP24S.HL,
AE_MULSP24S.LH, AE_MULSP24S.LL, AE_MULSQ32SP16S.H, AE_MULSQ32SP16S.L, AE_MULSQ32SP16U.H,
AE_MULSQ32SP16U.L, AE_MULSRFQ32SP24S.H [EP], AE_MULSRFQ32SP24S.L [EP], AE_MULSS56P24S.HH,
AE_MULSS56P24S.HL, AE_MULSS56P24S.LH, AE_MULSS56P24S.LL, AE_MULSSFP24S.HH.LL,
AE_MULSSFP24S.HL.LH, AE_MULSSP24S.HH.LL, AE_MULSSP24S.HL.LH, AE_MULZAAFP24S.HH.LL,
AE_MULZAAFP24S.HL.LH, AE_MULZAAFQ32SP16S.HH, AE_MULZAAFQ32SP16S.LH,
AE_MULZAAFQ32SP16S.LL, AE_MULZAAFQ32SP16U.HH, AE_MULZAAFQ32SP16U.LH,
AE_MULZAAFQ32SP16U.LL, AE_MULZAAP24S.HH.LL, AE_MULZAAP24S.HL.LH, AE_MULZAAQ32SP16S.HH,
AE_MULZAAQ32SP16S.LH, AE_MULZAAQ32SP16S.LL, AE_MULZAAQ32SP16U.HH,
AE_MULZAAQ32SP16U.LH, AE_MULZAAQ32SP16U.LL, AE_MULZASFP24S.HH.LL, AE_MULZASFP24S.HL.LH,
AE_MULZASFQ32SP16S.HH, AE_MULZASFQ32SP16S.LH, AE_MULZASFQ32SP16S.LL,
AE_MULZASFQ32SP16U.HH, AE_MULZASFQ32SP16U.LH, AE_MULZASFQ32SP16U.LL,
AE_MULZASP24S.HH.LL, AE_MULZASP24S.HL.LH, AE_MULZASQ32SP16S.HH, AE_MULZASQ32SP16S.LH,
AE_MULZASQ32SP16S.LL, AE_MULZASQ32SP16U.HH, AE_MULZASQ32SP16U.LH, AE_MULZASQ32SP16U.LL,

AE_MULZSAFP24S.HH.LL, AE_MULZSAFP24S.HL.LH, AE_MULZSAFQ32SP16S.HH, AE_MULZSAFQ32SP16S.LH, AE_MULZSAFQ32SP16S.LL, AE_MULZSAFQ32SP16U.HH, AE_MULZSAFQ32SP16U.LH, AE_MULZSAFQ32SP16U.LL, AE_MULZSAP24S.HH.LL, AE_MULZSAP24S.HL.LH, AE_MULZSAQ32SP16S.HH, AE_MULZSAQ32SP16S.LH, AE_MULZSAQ32SP16S.LL, AE_MULZSAQ32SP16U.HH, AE_MULZSAQ32SP16U.LH, AE_MULZSAQ32SP16U.LL, AE_MULZSSFP24S.HH.LL, AE_MULZSSFP24S.HL.LH, AE_MULZSSFQ32SP16S.HH, AE_MULZSSFQ32SP16S.LH, AE_MULZSSFQ32SP16S.LL, AE_MULZSSFQ32SP16U.HH, AE_MULZSSFQ32SP16U.LH, AE_MULZSSFQ32SP16U.LL, AE_MULZSSP24S.HH.LL, AE_MULZSSP24S.HL.LH, AE_MULZSSQ32SP16S.HH, AE_MULZSSQ32SP16S.LH, AE_MULZSSQ32SP16S.LL, AE_MULZSSQ32SP16U.HH, AE_MULZSSQ32SP16U.LH, AE_MULZSSQ32SP16U.LL, AE_NANDP48, AE_NANDQ56, AE_NEGP24, AE_NEGQ56, AE_NEGSP24S, AE_NEGSQ56S, AE_ORP48, AE_ORQ56, AE_ROUNDSP16ASYM, AE_ROUNDSP16Q48ASYM, AE_ROUNDSP16Q48SYM, AE_ROUNDSP16SYM, AE_ROUNDSP24Q48ASYM, AE_ROUNDSP24Q48SYM, AE_ROUNDSP32ASYM, AE_ROUNDSP32SYM, AE_SATQ48S, AE_SELP24.HH, AE_SELP24.HL, AE_SELP24.LH, AE_SELP24.LL, AE_SLLIP24, AE_SLLISP24S, AE_SLLSP24, AE_SLLSP24S, AE_SRAIP24, AE_SRASP24, AE_SRLIP24, AE_SRLSP24, AE_SUBP24, AE_SUBQ56, AE_SUBSP24S, AE_SUBSQ56S, AE_TRUNC16, AE_TRUNC16Q48X2, AE_TRUNCQ32, AE_XORP48, AE_XORQ56, AE_ZEROP48, AE_ZEROQ56

2.17.4 Core Instructions in slot0

ABS, ADD, ADDI, ADDMI, ADDX2, ADDX4, ADDX8, ALL4, ALL8, AND, ANDB, ANDBC, ANY4, ANY8, BALL, BANY, BBC, BBCI, BBS, BBSI, BEQ, BEQI, BEQZ, BF, BGE, BGEI, BGEU, BGEUI, BGEZ, BLT, BLTI, BLTU, BLTUI, BLTZ, BNALL, BNE, BNEI, BNEZ, BNONE, BT, CLAMPS, EXTUI, J, JX, L16SI, L16UI, L32I, L32R, L8UI, MAX, MAXU, MIN, MINU, MOVEQZ, MOVF, MOVGEZ, MOVI, MOVLTZ, MOVNEZ, MOVT, NEG, NOP, OR, ORB, ORBC, S16I, S32I, S8I, SEXT, SLL, SLLI, SRA, SRAI, SRC, SRL, SRLI, SSA8B, SSA8L, SSAI, SSL, SSR, SUB, SUBX2, SUBX4, SUBX8, XOR, XORB

上面只是类别，并不指详细的函数，因为同一类型的函数有很多种，比如ABS有AE_MAXABSSQ56S、AE_ABSSP24S、AE_ABSSQ56S等等。