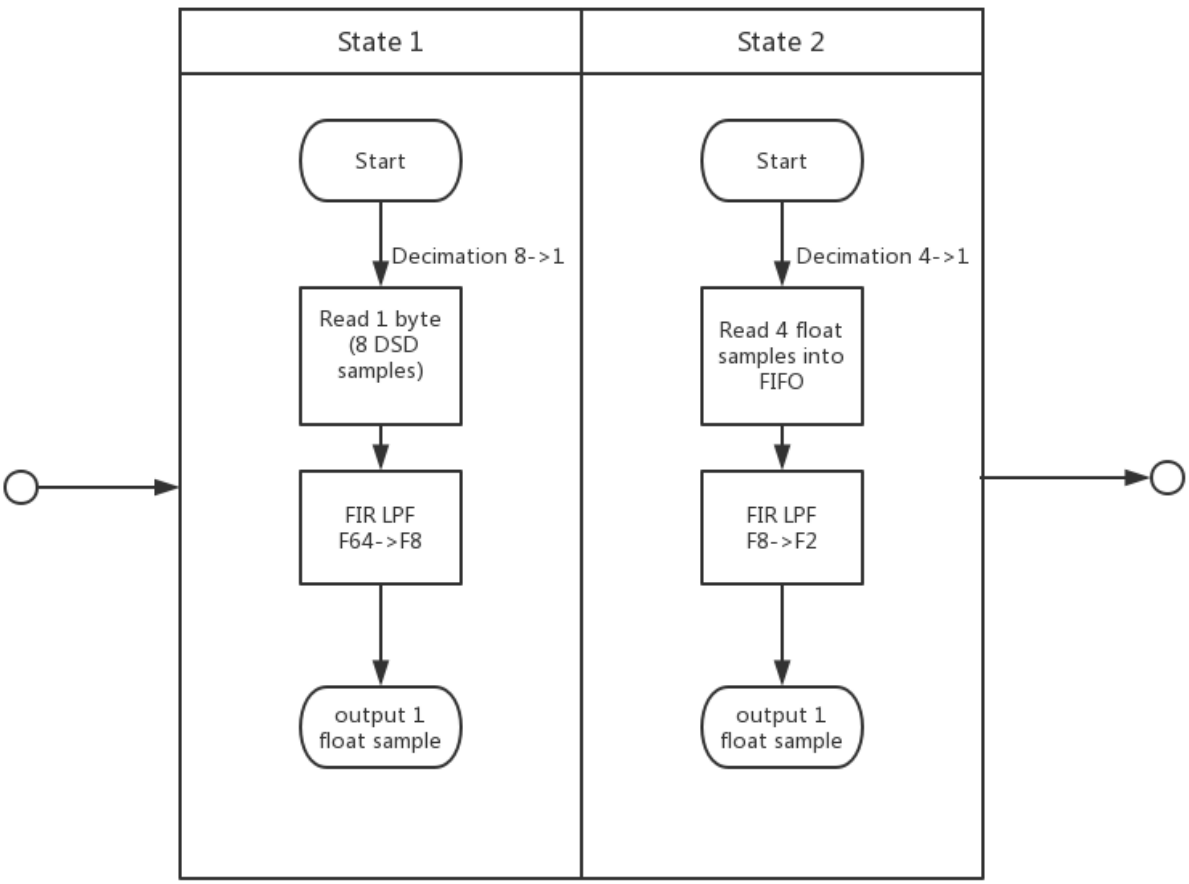
DSD2PCM 相关资源占用评估：

本文主要讨论 DSD2PCM Java 版本中的 DSD 转 PCM 算法的实现以及优化的策略，并针对原算法第二阶滤波的可优化方案进行了分析和讨论，提出基于 Half-band FIR filter 的可选方案。

在后续的优化策略中可以考虑在原有方案的基础上增加通过 SSE 指令集的对 FIR 滤波过程进行加速以及使用 FFT 对 FIR 阶数大于128 的滤波进行加速。除此之外，将浮点转定点再进行计算也是可以考虑的。

DSD2PCM 算法架构和表现

DSD2PCM，主要参考来自于 https://code.google.com/archive/p/dsd2pcm ，此版本为 C 语言版本，该作者后续拓展了 Java 版本，这两个版本在处理流程上有一定的区别，C 语言版本的处理只有下图中的 State1 前级处理，而 JAVA 版本的较为完善，包括后级处理 State2 ，如下图所示。



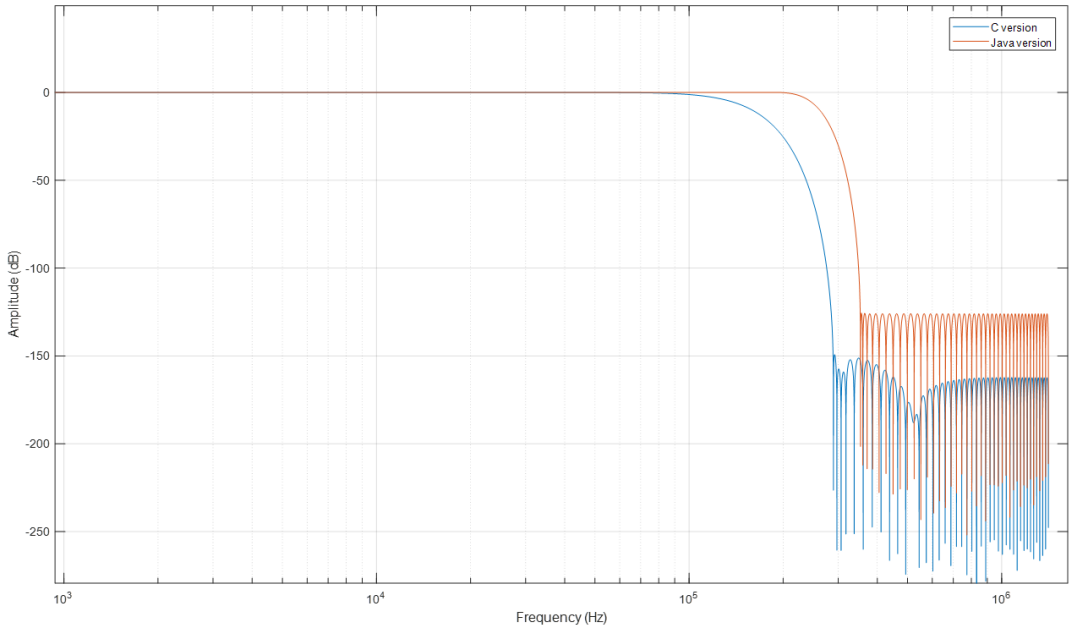
Java 版本和 C 语言版本 State1 前级滤波器的对比

Java 版本与 C 语言版本所不同的是在 State1 的基础上增加了 State2 的处理，就功能上来说，输出的 PCM 信号从原有的 352800 Hz（f8） 变为 352800 Hz 和 88200 Hz（f2）两路可选信号。

除了总体的处理流程有所变化之外，这两个版本中 State1 的滤波器参数也有了一些变化，本节将会这点分析这些区别，下面列出 C 语言和 Java 版本中所使用的 FIR 滤波器系数，并根据相关的频率响应做一个简单的分析：

```
  C_version_state1_coeffs[48]{
  0.09950731974056658,0.09562845727714668,0.08819647126516944,  0.07782552527068175,
  0.06534876523171299,0.05172629311427257,0.0379429484910187,   0.02490921351762261,
  0.0133774746265897,0.003883043418804416,-0.003284703416210726,-0.008080250212687497,
 -0.01067241812471033,-0.01139427235000863,-0.0106813877974587, -0.009007905078766049,
 -0.006828859761015335,-0.004535184322001496,-0.002425035959059578,-0.00069221870807907
08,
  0.0005700762133516592,0.001353838005269448,0.001713709169690937,0.001742046839472948,
  0.001545601648013235,0.001226696225277855,0.0008704322683580222,0.000538163620053564
9
,
  0.000266446345425276,7.002968738383528e-05,-5.279407053811266e-
05,-0.0001140625650874684,
 -0.0001304796361231895,    -0.0001189970287491285, -9.396247155265073e-05,
-6.577634378272832e-05,
 -4.07492895872535e-05,     -2.17407957554587e-05,  -9.163058931391722e-06,
-2.017460145032201e-06,
  1.249721855219005e-06,    2.166655190537392e-06,  1.930520892991082e-06,
 1.319400334374195e-06,
  7.410039764949091e-07,    3.423230509967409e-07,  1.244182214744588e-07,
 3.130441005359396e-08
  }
```

```
java_version_state1_coeffs[64]{
    0.09712411121659f, 0.09613438994044f, 0.09417884216316f, 0.09130441727307f,
         0.08757947648990f, 0.08309142055179f, 0.07794369263673f, 0.07225228745463f,
         0.06614191680338f, 0.05974199351302f, 0.05318259916599f, 0.04659059631228f,
         0.04008603356890f, 0.03377897290478f, 0.02776684382775f, 0.02213240062966f,
         0.01694232798846f, 0.01224650881275f, 0.00807793792573f, 0.00445323755944f,
         0.00137370697215f,-0.00117318019994f,-0.00321193033831f,-0.00477694265140f,
        -0.00591028841335f,-0.00665946056286f,-0.00707518873201f,-0.00720940203988f,
        -0.00711340642819f,-0.00683632603227f,-0.00642384017266f,-0.00591723006715f,
        -0.00535273320457f,-0.00476118922548f,-0.00416794965654f,-0.00359301524813f,
        -0.00305135909510f,-0.00255339111833f,-0.00210551956895f,-0.00171076760278f,
        -0.00136940723130f,-0.00107957856005f,-0.00083786862365f,-0.00063983084245f,
        -0.00048043272086f,-0.00035442550015f,-0.00025663481039f,-0.00018217573430f,
        -0.00012659899635f,-0.00008597726991f,-0.00005694188820f,-0.00003668060332f,
        -0.00002290670286f,-0.00001380895679f,-0.00000799057558f,-0.00000440385083f,
        -0.00000228567089f,-0.00000109760778f,-0.00000047286430f,-0.00000017129652f,
        -0.00000004282776f, 0.00000000119422f, 0.00000000949179f, 0.00000000747450f
}
```



在相关文档的描述中，C 版本的 state1 FIR filter 参数如下：

- 96 taps, sample rate at 44100*64
- flat response up to 48 kHz
- if you downsample afterwards by a factor of 8, the spectrum below 70 kHz is practically alias-free.
- stopband rejection is about 160 dB

Java 版本的 state1 FIR filter 参数如下：

- 128 taps, sample rate at 44100*64
- Passband is 0-24 kHz (ripples +/- 0.025 dB)
- Stopband starts at 176.4 kHz (rejection: 170 dB)

结合频响图可以验证，Java 版滤波器的通带大约到 176.4 kHz，C 版滤波器通则带为 0 ~ 70 kHz，阻带增益分别为 -130 dB，-150 dB。根据采样定理，经过 satae1 处理之后，这两个版本分别可输出 352.8 kHz，140 kHz 采样率的无失真有效信号。从这点来说，C 语言版本的设计指标与实际输出并不相符，不过作为前级处理，用于输出 96/88.2 kHz 的信号还是绰绰有余的。

结论：

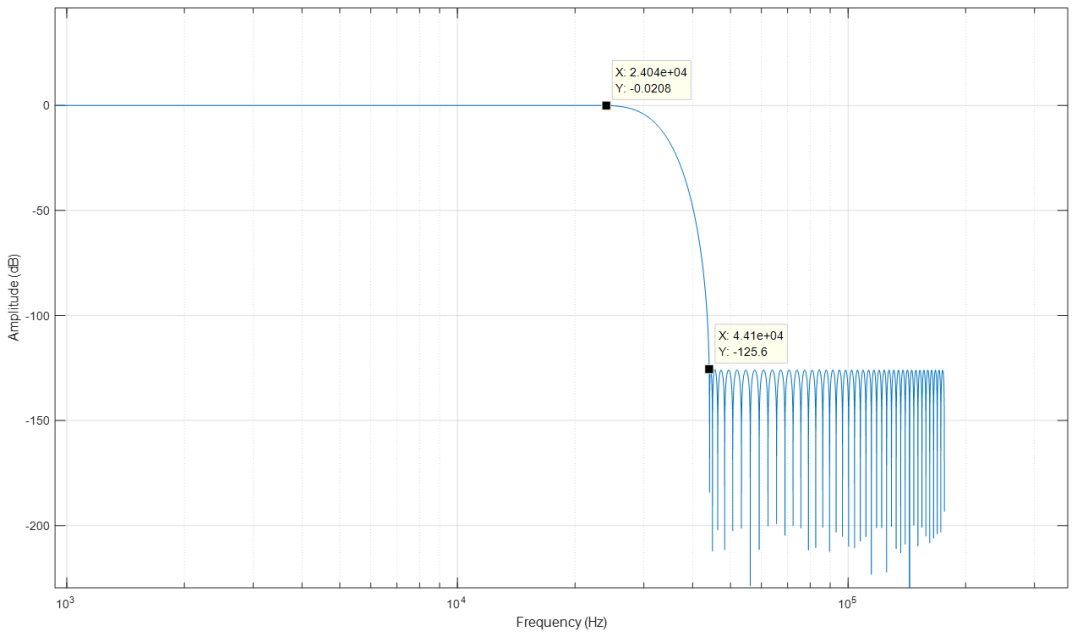
- C 语言版本和 Java 版本的 state1 滤波器所使用的系数和阶数均有所不同，导致实际滤波的效果也有所变化。
- 两版本分别可输出 140 kHz, 352.8 kHz 采样率的无失真有效信号（相对于原始 DSD 信号），阻带增益分别是 -150 dB，-130 dB，有后续处理时这两点应该纳入考虑范围

State2 滤波器分析

作为 Java 版本 DSD2PCM 中新添加的处理过程，主要负责将由 State1 输出的 352.8 kHz 信号进行滤波器处理，输出 88.2 kHz 信号。所使用的 FIR 滤波器参数如下，跟前级一样，都是标准 FIR 滤波器，利用系数对称的形式使用一半的系数以减少乘法运算过程，这属于优化部分，后续将会介绍。

```
state2_fir_coeffs[48]={
            0.17432985712158f, 0.15635873283656f, 0.12388722932786f, 0.08302746247746f,
        0.04109805784177f, 0.00496575740322f,-0.02040745815192f,-0.03285720617810f,
```

```
        -0.03316480848596f,-0.02448722392416f,-0.01128444209590f, 0.00192623920806f,
         0.01164802299008f, 0.01605827157540f, 0.01512886777062f, 0.01026408477035f,
         0.00363150734425f,-0.00258088650527f,-0.00676581108107f,-0.00821685345603f,
        -0.00713285331154f,-0.00436803832934f,-0.00105068213937f, 0.00179320983820f,
         0.00350400707295f, 0.00388856110053f, 0.00316403730329f, 0.00179777272355f,
         0.00031446525259f,-0.00086495262361f,-0.00151339001247f,-0.00160838203424f,
        -0.00128134808988f,-0.00073703173457f,-0.00017530374211f, 0.00026221396059f,
         0.00051228197090f, 0.00058124385617f, 0.00051918676923f, 0.00039047665991f,
         0.00025080092604f, 0.00013536634905f, 0.00005772301329f, 0.00001547370698f,
        -0.00000169056875f,-0.00000516205946f,-0.00000348297382f,-0.00000133768885f
}
```



滤波器阶数为 96阶，信号采样频率为 44100*8 Hz，输出信号为 44100 * 2 Hz，通带为 0~24 kHz (ripples +/- 0.02 dB ），阻带从 44.1 kHz (rejection: 126 dB) 开始，阻带增益约为 -125 dB。读者也许会对此有所疑问：为何输出 88.2 kHz 的信号，通带只设置到 24 kHz 而不是 44.1 kHz。这是由于 DSD 信号本身过采样的特性所致，建议在 30 kHz左右 设置一个低通滤波器，滤除过采样所产生的噪声**【此处补充相关的说明，尽量不与 halfband 相悖】**

综合 state1 以及 state2 两节滤波器的性能表现，Java 版本的 DSD2PCM 最终可以输出采样频率为 88.2 kHz的信号，其中 0~24 kHz 为无损信号，而从 44.1 kHz 开始则为阻带，增益约为 -125 dB。

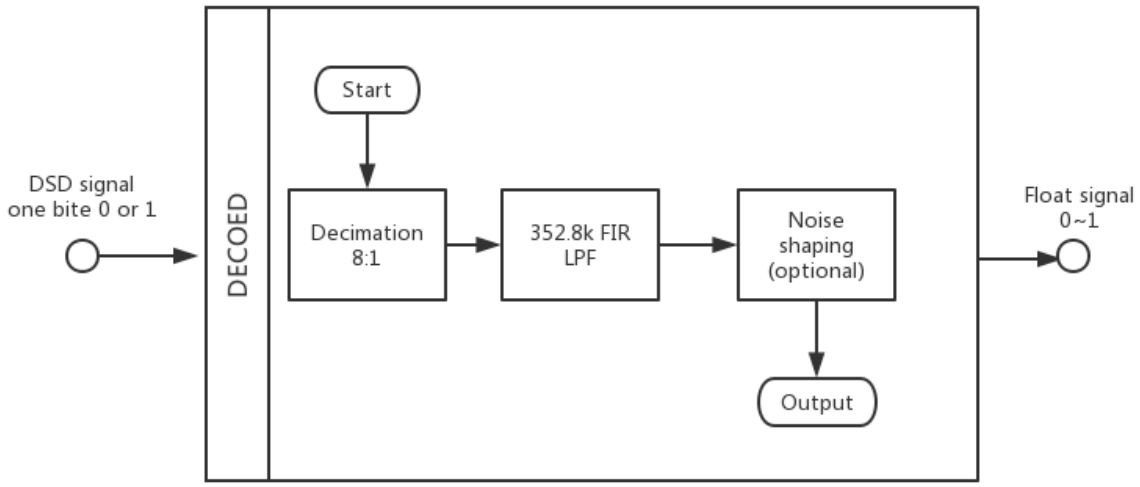
优化策略分析

DSD2PCM 使用了部分不少优化的技巧来对滤波算法进行了优化，减少算法的计算资源消耗，分析这些优化策略有助于理解设计者的设计意图，后续可以结合实际使用情况予以改进。

State1 分析

其中较为出彩的是 State1 中对用于 DSD 转 PCM 解码的 FIR 滤波器的优化：

从功能上来说，这个滤波器需要将 DSD 信号转化为 PCM 信号，而 DSD 信号在文件流中仅有一位（bit）表示，即输入信号为 0，1 。DSD 信号为 Delta-Sigma（ ΔΣ ）调制，解码需要用到累加器，DSD 编解码原理可参考**【链接】**，在这里使用 FIR 滤波器可以同时完成 DSD 信号的解码和滤波操作，最终输出浮点数据。State1 的处理流程如下图所示：



设计者在对 352.8 kHz FIR 低通滤波器进行优化时从以下几个方面进行：

首先根据 FIR 的定义式

$$y[n] = \sum_{i=0}^{N-1} b_i \cdot x[n-i] \tag{1}$$

标准 FIR 形式每输出一个样点，需要 N 个乘法和 N 个加法运算。现将其按照 8 个一组进行拆分 ，因此 N 需为 8 的倍数，注意选择 8 个一组作为拆分也是为了满足 Decimation 8:1 的抽取需求，从代码实现的角度来说，是因为一个字节为 8 位，如果是其他比值可以自行调整

标准 FIR 形式每输出一个样点，需要 N 个乘法和 N 个加法运算。现将其按照 8 个一组进行拆分 ，因此 FIR 节数 N 需为 8 的倍数，注意选择 8 个一组作为拆分也是为了满足 Decimation 8:1 的抽取需求，从代码实现的角度来说，一个字节为 8 位，8:1 抽取较为简洁，如果是其他比值可以自行调整。

$$y[n] = \sum_{i=0}^{N/8} \sum_{j=0}^{7} b_{8i+j} \cdot x[n-8i-j] \tag{2}$$

由于 DSD 信号的特殊性，其只有两种情况，0 或者1 ，因此内层每个滤波器系数对应输入信号的乘积 $b_i \cdot x[n]$ 只有 $2^8 = 256$ 种情况，（可以将 x 看成是一个 大小为 8 的 FIFO，每集齐 8 个点计算一次滤波器输出的值，DSD 输入只有 0 或 1，集齐 8 个点时 FIFO 总共有 256 种情况，制作表格时计算的是 8 个滤波器系数与 FIFO 中的值按照内层循环的计算公式计算后的总和，因此表格的大小为 $N/8 * 256$）这些数值可以提前计算好并写入表格中，通过查表避免乘法运算，最终每输出一个样点，理论上仅需要$N/8$ 个加法运算 ，极大地提高了运算的效率。

其次，对比设计者实际的代码可以发现，设计者在计算查找表时，使用的 FIR 滤波器节数 N 是标准 FIR 滤波器系数个数的一半，这是为了利用 FIR 滤波器系数对称的性质（state2 分析中将会介绍），减少一半的乘法运算。

虽然在最后合成样点的时候需要查询两次表格中的提前算好的乘法数据（对称）进行合成，但是相应的外层循环也相应地少了一半，因此整个过程中，加法运算总数与不进行拆分时是一样的: $ADD = N/8 * 2$ ，*N = half_order* 。通过这样的优化后，可以将查找表的数据占用空间减少一半，由此看来设计者在算法设计层面已经将代码优化到了极致，如果还想对 State1 的算法进行优化可以考虑从指令集的角度入手，加速外层 $N/8$ 个加法运算的过程，如

SSE2 指令集可以并行计算两组浮点数据间的运算。

资源占用分析

查找表大小公式：

在利用了系数对称性质后，查找表大小可以减少一半 ：

$$Table_{size} = N/8 * 256 * 4, N = halforder \tag{3}$$

其中每个浮点值占用 4 bytes，以 96 节 FIR 滤波器为例，生成的查找表大小约为 6k bytes。

FIFO buffer 长度：

长度 N 至少为 full orer

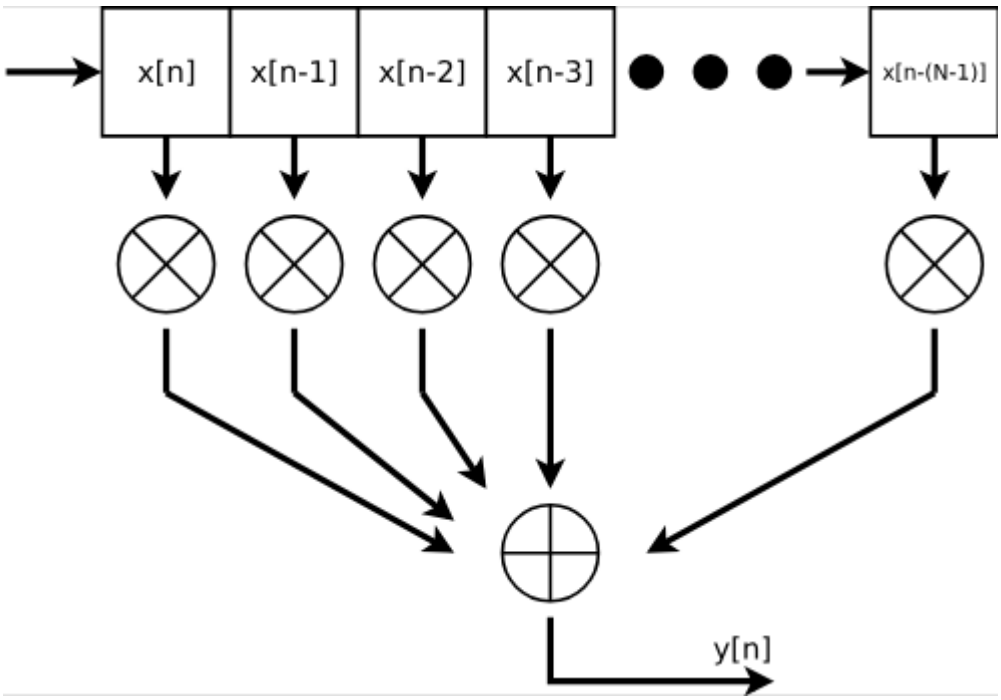
计算资源：

每计算一个样点值，需要 $N/8$, *N = full_order* 个加法运算（不包含数据存取）

【补充复现代码】

State2 分析

在 State2 中所使用的 FIR 滤波器则使用了较为常规的优化手段，即利用系数对称，减少乘法运算的数量，在 state1 中也可以找到，以下可以作为相应的补充：

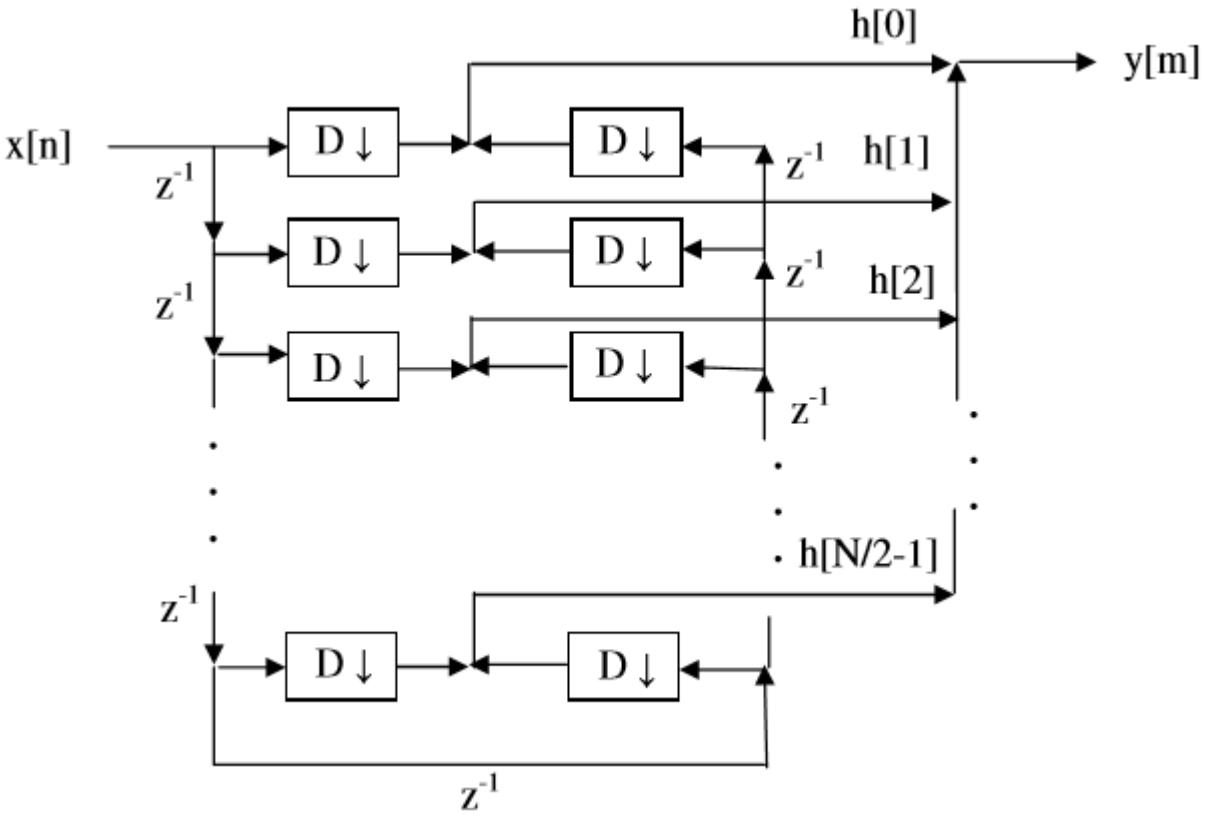


上图即为公式 (1) 的实现，对于 FIR 滤波器来说，其系数是关于中心对称的：

$$\begin{cases} b_0 = b_{N-1}, b_1 = b_{N-2}, \ldots, b_{N/2-1} = b_{N/2}, N = enen \\ b_0 = b_{N-1}, b_1 = b_{N-2}, \ldots, b_{(N-3)/2} = b_{(N+1)/2}, b_{(N-1)/2}, N = odd \end{cases} \tag{4}$$

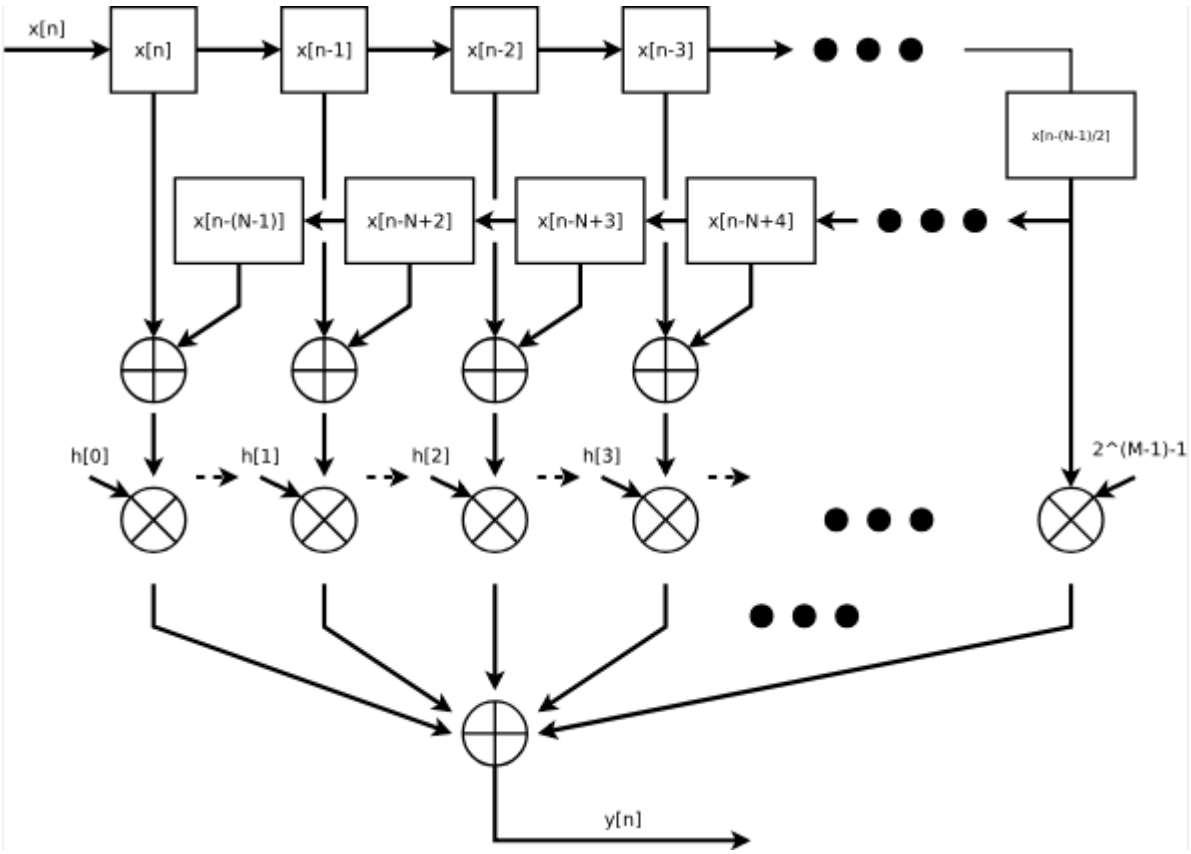
因此可以将上图进行如下转换，n = even：

$$y(n) = \sum_{k=0}^{N/2-1} b_k \cdot [x(n-k) + x(n-N+1+k)] \tag{5}$$



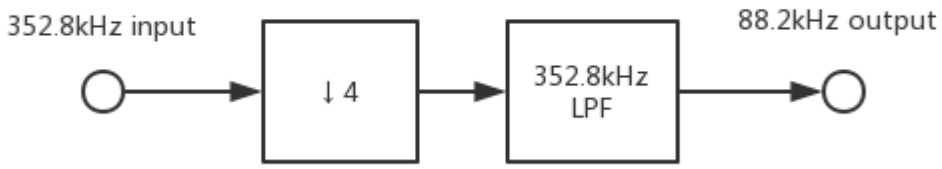
同理，n = odd 时：

$$y(n) = b_{(N-1)/2} \cdot x_{n-(N-1)/2} + \sum_{k=0}^{N-3/2} b_k \cdot [x(n-k) + x(n-N+1+k)] \tag{6}$$



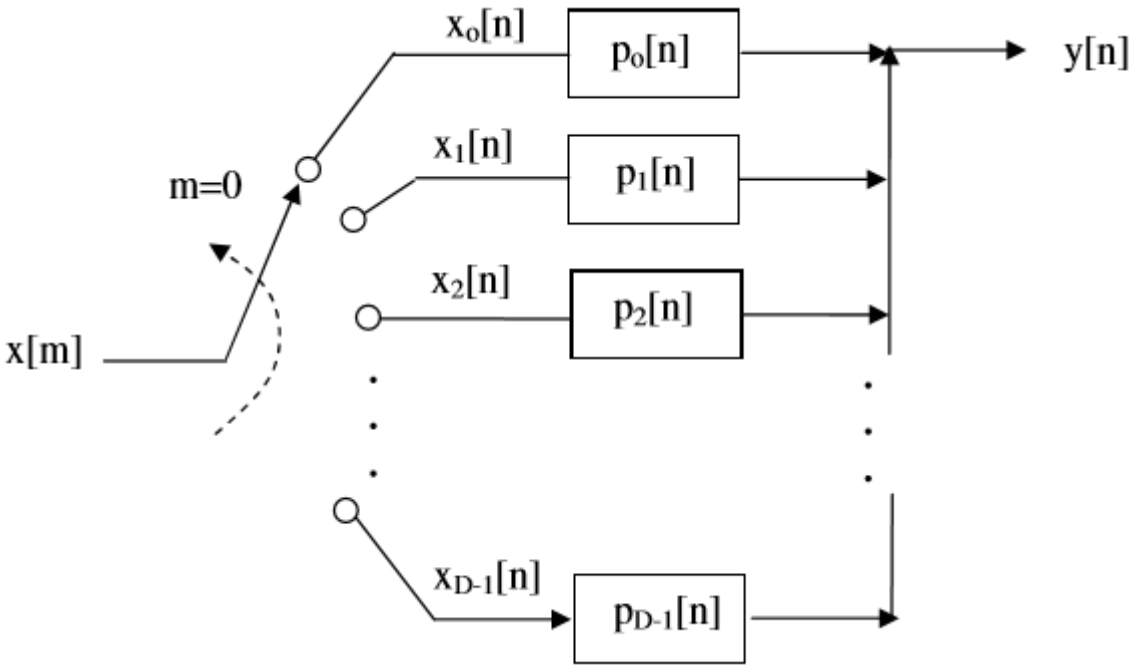
利用系数对称的性质，每计算一个点，需要的 N/2 个乘法运算，以及 N 个加法运算，N = even full order。FIFO buffer 长度至少为 full order。与 State1 所不同的是由于输入、输出均为浮点，无法通过查表的方式进行优化。

除此之外，state2 中还集成了 D=4:1 的抽取器（decimation），不过抽取操作是等价于在滤波之后进行的：



实际上在 state2 的实现中，通过将每次的输入都存到一个长度至少为 N = nCoeffs 的 FIFO buffer 中，经过每 D= 4 个点计算一次滤波器的输出值，因此滤波器的采样频率需要与原输入信号的频率是一致的。这种结构可以应用 FIR 系数对称的性质减少一半的乘法运算，由于每集齐 D 个样点值才会进行一次 FIR 滤波操作，所以通过 FIR 滤波的点数为输入信号的 1/D 。

如果在滤波之前先进行抽取，同样可以减少通过滤波器的样点数，不过这种实现下，滤波器的频率需要与输出信号的频率一致，这类型的结构称为 polyphase structure。需要注意的是，polyphase 结构系数并不会完全满足系数对称的形式，另一个不同点是 state2 中滤波器的采样频率与输入信号一致，而 polyphase structure 的采样频率与输出信号频率系统。 polyphase structure 可以如下所示：



在后续的优化中可以考虑将 State2 的滤波器换为 polyphase 结构。

总结：

- State2 利用 FIR 滤波器系数对称的性质，将处理每一个 FIR 输入所需要的乘法运算降低一半。 $N_{mul} = N_{fullorder}/2$
- 在 down samplerate 操作中，是按照 "经过滤波后再按比例抽取输出信号" 的处理思路来进行，不过使用 FIFO buffer 可以在经过 D 点后才调用 FIR 进行滤波，变相降低需要通过 FIR 滤波的样点数，抽取比例为 D:1，

$D = fs_{in}/fs_{out}$，时，需要通过 FIR 滤波处理的点数为 $N_{filter} = N_{total}/D$ ，这个思路与 polyphase 结构相一致，不同之处在于此处 FIR 滤波器的采样频率需要与输入频率相一致，而 polyphase 结构则等于输出频率。

【上图中 lpf 的采样频率为 352.8kHz 通带为 0~24 kHz，阻带从 44.1 kHz 开始。对 352.8kHz 信号进行 4:1 的抽取时，会发生混叠，由奈奎斯特定理可知，当输出为 88.2 kHz 的信号时，44.1 kHz 以上的信号会发生混叠并无法保证其有效性，【待补充相关论证】

由此可得只要滤波器能够保证将混叠的部分滤除，先抽取再滤波或者是先滤波再抽取两者的效果都是一致的，先抽取再滤波则可以减少需要通过 FIR 处理的样点数，进一步降低计算需求。】

对于 State2 FIR LPF 的优化改进

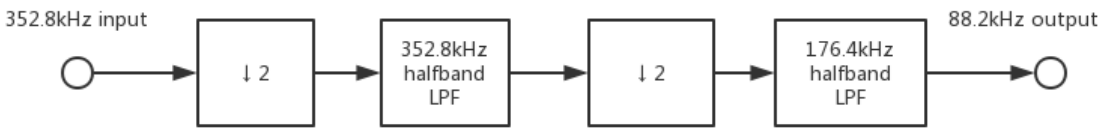
通过前面的分析我们可以知道，在 State2 中的 LPF 设计指标如下：

- 阶数为 96 阶，其输入为 352.8 kHz 采样频率的浮点信号，输出为 88.2 kHz 采样频率的信号，通带为 0~24 kHz (ripples +/- 0.02 dB ），阻带从 44.1 kHz 开始，阻带增益约为 -125 dB。
- 使用的架构为系数对称 FIR 结构，每输出一个点，需要用到 48 个乘法运算以及 96 个加法运算。

通过一系列的前置研究，笔者发现 FIR half band 结构有在占用较少资源的情况下，满足设计指标的潜质，接下来会围绕这个点进行研究和评估。

设计框架：

使用两节 Half band FIR，第一节为 96 阶，第二节为 48阶 ，对应的采样频率分别为 352.8 kHz 以及 176.4 kHz。



根据 halfband FIR filter 的特性【通带为 fs / 4】，第一节 需要 24 个乘法器，第二节需要 12 个乘法器，总数小于原方案的 48 个乘法器需求。

设计指标：与 state2 中的相一致，

`b = firhalfband(n,dev,'dev')` designs an Nth-order lowpass halfband filter with an equiripple characteristic. Input argument `dev` sets the value for the maximum passband and stopband ripple allowed.