分类号：_____ 密级：_____

UDC：_____ 编号：_____

学 位 论 文

MP3 音频解码器的 FPGA 实现

张晓境

指导教师姓名：田学民 副教授 河北工业大学

申请学位级别：硕 士 学科、专业名称：物理电子学

论文提交日期：2012 年 11 月 论文答辩日期: 2012 年 12 月

学位授予单位： 河北工业大学

答辩委员会主席：_____

评 阅 人：_____

2012 年 11 月

Dissertation/Thesis Submitted to

Hebei University of Technology

for

The Master Degree of

Physical Electronics

The Implementation of FPGA for MP3 Audio Decoder

by

Zhang Xiaojing

Supervisor: Prof. Tian Xuemin

DEC 2012

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，进行研究工作所取得的成果。除文中已经注明引用的内容外，本学位论文不包含任何他人或集体已经发表的作品内容，也不包含本人为获得其他学位而使用过的材料。对本论文所涉及的研究工作做出贡献的其他个人或集体，均已在文中以明确方式标明。本学位论文原创性声明的法律责任由本人承担。

学位论文作者签名：张晓境　　日期：2022年11月5日

关于学位论文版权使用授权的说明

本人完全了解河北工业大学关于收集、保存、使用学位论文的以下规定：学校有权采用影印、缩印、扫描、数字化或其它手段保存论文；学校有权提供本学位论文全文或者部分内容的阅览服务；学校有权将学位论文的全部或部分内容编入有关数据库进行检索、交流；学校有权向国家有关部门或者机构送交论文的复印件和电子版。

（保密的学位论文在解密后适用本授权说明）

学位论文作者签名：张晓境　　日期：2022年11月5日

导　师　签　名：　　日期：2022年11月5日

MP3 音频解码器的 FPGA 实现

摘 要

如今 MP3 已经成为了最流行的音乐格式，因为与其他格式的音乐相比较，MP3 格式除了在其音质、压缩性方面有优势，在其目前的传播广泛性和互联网上的免费下载方面，也占有绝对的优势地位。所以成为了应用最广泛的移动设备。而 MP3 播放器的性能，主要取决于 MP3 内部的解码器，因此我们以 MP3 解码器为核心，主要研究了 MP3 解码电路的设计方法，以及 FPGA 的实现。

本文主要介绍了 MP3 文件格式和解码原理，在基于 FPGA 的硬件开发平台上，设计和实现了 MP3 的解码器的功能。在基于 Altera 公司的 Cyclone Ⅱ系列的 EP2C35F672C6 开发平台，利用 Verilog HDL 硬件描述语言对 MP3 解码系统中的各个模块进行硬件设计，是本文的一大特色。根据解码过程将总体设计进行模块划分，主要分为：帧边信息解码模块，比例因子模块，Huffman 解码模块，反量化模块，重排列模块，立体声模块，混叠重建模块，IMDCT 模块等。

1. 为了加快解码过程，对反量化模块的采用了改进的除 8 查表法，减少了对缓存区的访问次数，很大程度上提高了运算的速度和精度。

2. 在 IMDCT 模块中，IMDCT（改进的离散余弦反变换）是一个计算量非常大的过程，占用的时间最长，因此对结构进行优化是十分有必要的。本论文将 IMDCT 的三个步骤融合在一起进行硬件实现，减少了对内存的访问次数，又节省了空间。

3. 用 Verilog HDL 硬件描述语言实现各模块的功能，并对主要模块进行了经过综合、仿真，分析了仿真结果，实现了 MP3 解码器的功能。

经过仿真，本设计实现了 MP3 解码器的基本功能，并且加快了解码的速度，具有广泛的应用前景。

关键字：MP3 解码器，Verilog HDL，FPGA

The Implementation of FPGA for MP3 Audio Decoder

ABSTRACT

Nowadays MP3 has become the most popular music format, comparing with other format, MP3 format except in its sound quality, compressibility has an advantage,in current propagation universality and the Internet free download aspect, also has an absolute advantage.So MP3 become the most widely used mobile device. And the core of a MP3 is decoder.So we focused on the MP3 decoder, studied the MP3 decoding circuit design and the implementation on FPGA.

This paper mainly introduced the MP3 file format and decoding principle, based on FPGA hardware development platform, design and implementation of the MP3 decoder. Based on Altera company's Cyclone Ⅱ series of EP2C35F672C6 development platform, use Verilog HDL hardware description language to MP3 decoding system of each module in hardware design, and it is a great characteristic of this paper. According to the overall design process of decoding module partition, mainly divides into: side_information decoding module, scale_factor module, Huffman decoding module, inverse quantization module, relining module, stereo module, aliasing reconstruction module, IMDCT module, etc.

1. In order to speed up the decoding, the inverse quantization module adopted improved look-up table method, namely except 8 look-up table method, only basic lookup table about one 8 of the storage space, and reduced the number of the cache access, to a great extent, improved the operation speed and precision.

2. In IMDCT modules, IMDCT (Inverse Modified Discrete Cosine Transform) is a very large amount of calculation, and the time is the longest, so the optimized algorithm is very necessary. This paper adopted the easy to hardware implementation of the recursive algorithm, and the improved algorithm is derived the steps, combined with the algorithm achieve the aim of the structural optimization.

3. Use Verilog HDL hardware description language to realize the function of each module, and the main modules after the comprehensive, simulation, analysis of the simulation results, realized the MP3 decoder function.

Through simulation, the design to realize the basic function of the MP3 decoder, and accelerate the decoding speed.So it has a broad prospect of application.

KEY WORDS: MP3 decoder, Verilog HDL,FPGA

目 录

第一章 绪论

§1-1 研究背景

随着计算机的普及，网络已经在人们的生活中占据了很大一部分，互联网上有大量的 MP3 格式的歌曲可以供人们免费下载，而且 MP3 格式有很好的高压缩比和良好音质，是当前最受欢迎的格式，MP3 播放器在市场是占据着绝对的优势地位，嵌入到了各种各样的设备中，应用性十分广泛。

现在数字音频压缩技术发展十分迅速，数字音频信号处理在很多领域都在发挥重要的作用，所以数据压缩是一个十分值得研究的内容。MPEG 是目前应用最广泛的标准，主要有以下五个，MPEG-1、MPEG-2、MPEG-4、MPEG-7 及 MPEG-21 等。我们通常所说的 MPEG-X 版本，就是由 ISO 所制定而发布的视频、音频、数据的压缩标准[1]。MPEG 标准的压缩技术主要利用的是帧间压缩编码技术，这种技术可以减小编码所需时间提高编码效率，综合利用这些技术可以最大程度的增强压缩特性。

1.MPEG-1

MPEG Audio 标准分三个层次 Layer 1、Layer2、Layer3，三个层的采样频率为 32KHz、44.1KHz、48KHz。

层 1(Layer 1）：编码简单，用于数字盒式录音磁带。

层 2(Layer 2）：算法复杂度中等，用于数字音频广播（DAB）和 VCD 等。

层 3(Layer 3）：编码复杂，用于互联网上的高质量声音的传输，如 MP3 音乐压缩 10 倍。

- MPEG-1 audio layer 1

所需频宽：384kbps 压缩率 4:1

特性：双声道，编码简单，用于数字磁带中。

优点：编码效率高，音质有很大的提高。

- MUSICAM(MPEG-1 audio layer 2，即 MP2)

所需频宽：256～192kbps

压缩率 8:1--6:1

特性：算法的复杂度和音质有理想的折中

- MP3(MPEG-1 audio layer 3)

所需频宽：128～112kbps

压缩率 12:1--10:1

Layer 3 就是我们常说的 MP3，它的音质决于它的解码器的质量。MP3 的采样频率一般采用的 44.1KHz，另外也有的采用 32KHz 和 48KHz 的。

2.MPEG-2

MPEG-2 制定于 1994 年，设计目标是高级工业标准的图象质量以及更高的传输率。MPEG-2 所能提供的传输率在 3-10Mbits/sec 间，其在 NTSC 制式下的分辨率可达 720×486，MPEG-2 也可提供并能够提供广播级的视像和 CD 级的音质。MPEG-2 的音频编码可提供左右中及两个环绕声道，以及一个加重低音声道，和多达 7 个伴音声道（DVD 可有 8 种语言配音的原因）。由于 MPEG-2 在设计时的巧妙处理，使得大多数 MPEG-2 解码器也可播放 MPEG-1 格式的数据，如 VCD。

MPEG-2 的优点就是能在很大的程度上改变压缩比例，带来高清晰的画面质量，而且可以满足不

同带宽的要求。但是对于使用的用户来说，由于市面上电视机的分辨率比较低，在电视上并不能明显地感觉到，MPEG-2 所带来的高清晰度画面质量，但是我们可以深刻地感受到，它的音质，尤其是低音方面的出色发挥[2]。

同时，由于 MPEG-2 的出色性能表现，已能适用于 HDTV，使得原打算为 HDTV 设计的 MPEG-3，还没出世就被抛弃了。（MPEG-3 要求传输速率在 20Mbits/sec-40Mbits/sec 间，但这将使画面有轻度扭曲）。除了做为 DVD 的指定标准外，MPEG-2 还可用于为广播，有线电视网，电缆网络以及卫星直播(Direct Broad cast Satellite）提供广播级的数字视频。

3.MPEG-4

数字信息的解压缩，是一个很重要的研究课题，当前 MPEG-X 系列的标准已经非常完整，基于 MPEG 音频标准的 FPGA 实现的研究是一个很值得研究的领域。本研究以提高解码速度，同时减少电路设计的面积为原则，优化解码算法，最终实现 MP3 解码功能。

MPEG 专家组的专家们正在为 MPEG-4 的制定努力工作。MPEG-4 标准主要应用于视像电话，视像电子邮件和电子新闻等，其传输速率要求较低，在 4800-64000bits/sec 之间，分辨率 176X144。MPEG-4 利用很窄的带宽，通过帧重建技术，压缩和传输数据，以求以最少的数据获得最佳的图象质量。

与 MPEG-1 和 MPEG-2 相比，MPEG-4 的特点是其更适于交互服务以及远程监控。MPEG-4 是第一个使你由被动变为主动（不再只是观看，允许你加入其中，即有交互性）的动态图象标准；它的另一个特点是其综合性；从根源上说，MPEG-4 试图将自然物体与人造物体相溶合（视觉效果意义上的）[3]。MPEG-4 的设计目标还有更广的适应性和可扩展性。

至于大家熟悉的 MP3 其实是 MPEG-1 Layer 3 的音频数据压缩技术，简称 MP3。目前 MP3 因其良好的音质、复杂度与压缩比的完美折中，它的市场广泛，在长时间内都会是首选的数字音频压缩技术,并保持其主流地位。各种各样的 MP3 音乐在网上是开放和免费的，在互联网很受欢迎,完全进入移动电话、电子词典等手持设备领域,有嵌入式 MP3 解码功能需求, MP3 解码器作为一个消费者电子产品非常受欢迎。

数字信息的压缩、解压缩电路是 IC 设计的一个重要领域，当前 MPEG 系列的音频标准已经十分成熟,设计 MPEG 音频解码芯片不存在大的技术风险，因而对 MPEG 音频解码算法的硬件实现的研究具有重要的现实意义。本研究以减少电路面积为设计原则，实现 MP3 解码功能。

§1-2 课题研究的意义

MP3 的解码系统只要有两种，软件解码和硬件解码，即对 MP3 软件解码系统的算法优化研究和 MP3 硬件解码系统的设计研究。

软件解码方面解码系统的解码算法非常复杂，所以可以对于其解码系统中的一些流程进行算法上的优化，以达到减小运算量和系统资源消耗的目的。

现阶段一般是采用 DSP 或 RISC 现 MP3 格式的解码。主要有三种实现方法，

(1)MP3 解码芯片，再加上外围电路；

优点：功耗低，速度最快。

缺点:功能比较单一，只是针对解码，灵活性差，性能低，不易升级。

(2)以 DSP 处理器作为主要模块，加上外围器件来实现；

优点：多种多样的功能，具有多种应用功能，很容易进行升级。

缺点：内存和运算量大，性能低，成本高。

(3)CPU/RISC 和其它硬件加速模块的 SOC（系统级芯片）设计加上外围器件实现。

优点：功耗小，价格低，支持多功能软件的升级。

缺点：还没有完全成熟。

从以上比较中，我们选择第三种实现方法为研究方向。第三种方案一般都是基于 FPGA(现场可编程门阵列)实现的，而 FPGA 最显著的一点就是不仅开发周期短而且实时性好，还很容易对它进行升级，是一个值得研究的领域。

§1-3 课题研究的主要内容

本文的主要研究内容是 MP3 解码过程中的解码算法，以 FPGA 器件实现解码功能为目标，提出了 MP3 解码器的系统结构。用 Verilog HDL 语言进行编程设计 MP3 解码器的各个模块，最后调用 modelsim 软件进行 RTL 级仿真。

MP3 音频压缩技术的编解码过程如图 1.1 所示，模拟的信号经过 A/ D 转换器、采样变换后成为 PCM 数字信号，再经过 MP3 编码器，利用 MP3 编码技术对 PCM 数字信号进行压缩，就变成了 MP3 格式文件， MP3 音频信号经音频解码器解码后，还原出初始的 PCM 数字信号，把恢复出的 PCM 信号传输到到 D\A 数模转换器就可以恢复成最开始的的模拟声音信号[4]。

MP3 音频格式的的编解码过程如图 1.1 所示

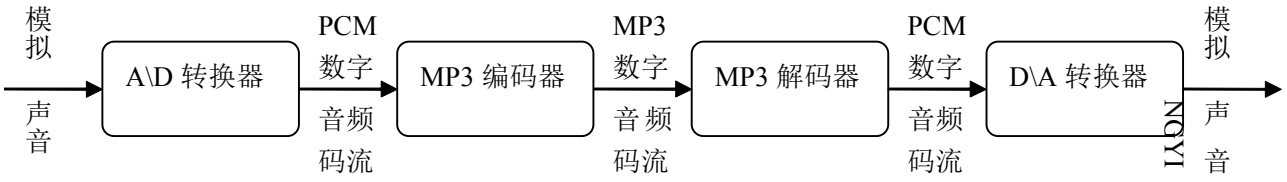

图 1.1 MP3 音频编解码过程示意图

Fig.1.1 The schematic diagram of MP3 audio decoding process

MP3 解码器要正常播放那么还原出的 PCM 音频信号就要满足实时性的要求，MP3 解码示意图如下图 1.2 所示，也就是每帧数据的解码时间要小于一个极限，才能保证声音播放的连续性。大多数 MP3 格式文件的采样频率为 44.1KHz，那么采样形成的 PCM 值所需要的时间为 1/44.1（ms）。如图 1.2(a)所示，采样 1152 个 PCM 数据构成一帧 MP3 解码所需要的时间是 1152/44.1=26(ms)。MP3 格式的音乐首先经过解码还原成 PCM 数字音频，然后必须按照采样时的速率把 PCM 数据输送回数模转换芯片，经过数模转换器后才能进行播放声音，所以播放一帧解码结构所需要的时间与采样时间相等[5] 。

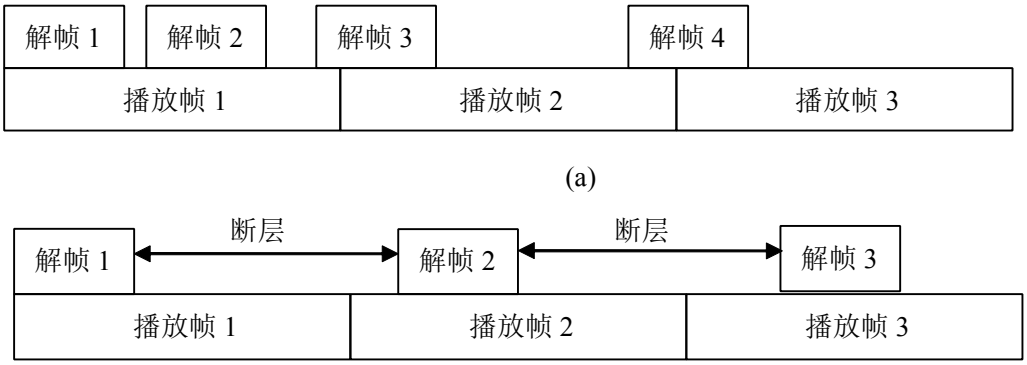

图 1.2 MP3 实时解码示意图

Fig.1.2The schematic diagram of MP3 real-time decoding

首先对解帧 1 进行解码，完成解码后就开始播放帧 1 的过程，当解帧 1 的结果播放完成后就要接着播放解帧 2 的 PCM 解码值，这时候只有在帧 2 已经完成了解码时，音频信号才能连续的播放出来，不然就会如图 1.2(b)所示的一样,解帧 1 的解码结果播放完成后，解帧 2 的还正在进行解码，播放帧 2 的阶段就不能连续进行下去，也就是说当帧 2 的解码时长比帧 1 的播放时长还要长时，我们听到的是断断续续的声音信号。因此为了保证持续地播放声音，就对解码设备的解码时长有硬性的要求，即解码时长必须要小于播放时长[6]。因此上对于 44.1KHz 采样频率的 MP3 格式文件，解码设备的解码时长不能超过 26ms。

考虑到解码器的实时性，我们在研究 MP3 硬件解码器时，如果采用左声道和右声道同时解码方法，可以提高解码速度，缩减所需要的时间。采用各模块分时复用数据总线的方法，以求节约存储空间，节省电路的面积。

本论文的结构安排如下：

第一章论述 MPEG 系列标准的具体内容，以及 MP3 解码器广泛应用和研究的背景、意义。

第二章分析 MP3 解码的原理，解码的流程以及研究了各子模块的算法。

第三章介绍了解码器的总体结构设计，采用内部共享存储单元，复用地址和数据总线的方法，以减少对存储器访问量，提供解码速度。

第四章主要内容是把 MP3 解码器内部划分成模块，并对各个子模块的具体流程进行分析，画出流程图，用 Verilog HDL 语言对各个模块的功能进行设计。

第五章用 Quartus Ⅱ软件对模块完成仿真，以 Cyclone Ⅱ系列的 EP2C35F672C6 的 FPGA 做为开发平台，实现 MP3 解码器的功能。此外还对仿真出的波形作出分析。

第六章对本设计进行了总结。

第二章 MP3 解码原理及算法研究

§2-1 MP3 解码原理

MP3 解码过程就是编码的反过程，MP3 的解码总体上可分为 9 个过程：比特流分解、Huffman 解码、反量化解码、重排列、立体声处理、混合重建、IMDCT 变换、频率反转补偿、子带合成滤波。解码流程如图 2.1 所示：

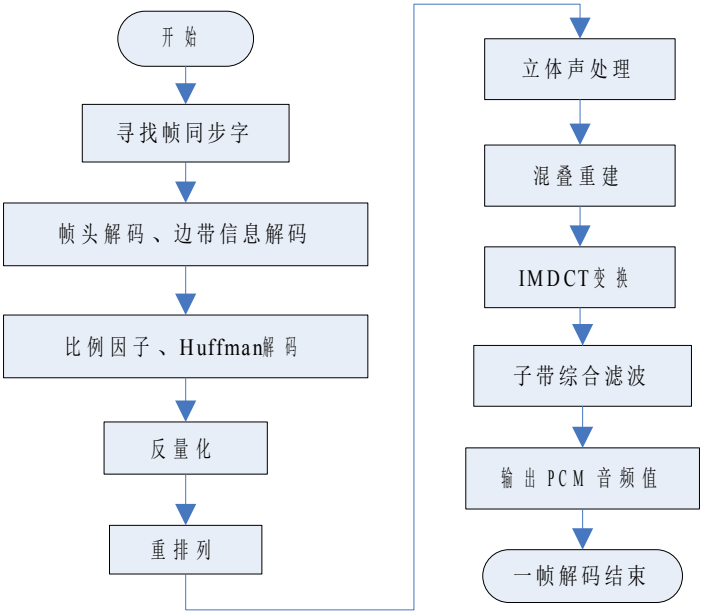

图 2.1 MP3 解码流程

Fig.2.1 Chart of Mp3 decoding flow

输入数据是比特流数据，流程图的最后产生的是 PCM 数据，大致分 3 个阶段：

1.经过语法和错误校验，将产生的哈曼数据送入霍夫曼解码器，同时用比例因子解码器把比例因子提取出来。2.把比例因子和 Huffman 解压出来的数据送入反量化器产生 MDCT 数据，经过重新编排器，把产生的 MDCT 数据送入联合立体声解码器。3.联合立体声解码器产生的数据经过混叠重建、IMDCT 模块、子带综合滤波器，最终恢复出 PCM 的数据。

压缩处理 PCM 信号时，组成的数据帧的长度是一定的的，帧是 MP3 文件的组成单元，每帧包含 1152 个采样值[7]。在解码过程中，把帧中的参量提取出来，就可以恢复出 PCM 的采样值。一个数据帧有两个粒度 granule 0 和 granule 1，每个粒度分别有 576 个 PCM 采样值。下面将介绍解码的大致过程。

1. 同步以及帧头信息的读取

帧是 MP3 码流的最小组成单位，同步信息有 12 位，包含于每帧的帧头中。解码时的首先要做的就是找到同步信息，时解码器与输入数据流同步，解码开始时就能利用数据中的 12 比特的同步字来完成。得到同步信息后，就可以接着提出帧头信息中的其他参量。

2. 主数据的读取

帧格式中主数据所包含的参数主要是比例因子,霍夫曼码字和一些附加数据。在 MP3 编码过程采用了比特池技术,当前帧的数据可以放在前面的帧中，以至于解码时当前帧的主数据不一定完全在当前帧,因此为了明确指出数据开始的位置，可以设定一个指针，用来确定主数据开始的位置在当前帧还是其他帧中[8]。主数据中的数据都有都是按照规定的格式分布的，是不会改变的。

3. Huffman 解码和反量化

Huffman 解码就是通过 Huffman 解码模块访问主数据 RAM 单元，取出所需要的声道信息。经过查找 Huffman 码表得到所需要的结果，主数据中的两个粒度在不同的区域需要不同的 Huffman 码表进行解码，解码后的结果通过总线存放在左/右声道各自的 RAM 中。

接下来就是反量化的过程，模块从左/右声道中，取出 Huffman 解码后的数据，运用反量化的公式进行计算，在此过程中对于不同的块类型的数据所使用的反量化公式也有所不同。反量化过程实际上就是通过公式恢复出每条频率线的真实数值。

4. 重排序和混叠重建

反量化后进行的就是重排列和混叠重建的过程，MP3 编码时的离散余弦变换过程中产生的频谱值，长块类型的数据排列顺序是子带、频率，短块类型的数据排列顺序是子带、窗、频率。编码时为了提高效率重新排列了短块中的数据，重排顺序是子带、频率、窗[9]。

所以解码过程中的重排列实际就是重排短块中的顺序，恢复成原来的子带、频率、窗的顺序。同样的原理，由于编码时对长块进行了去混叠处理，编码时，就要进行混叠重建，以便完整无误地还原原来的信号。

混合重建只是重建长块类型数据，对短块类型数据不处理，需要混叠重建的是长块数据和混合块中的长块部分。这部分在研究算法的过程中可以利用频率线零值区特性，不需要再进行蝶形计算就可以直接给出 0 值结果，减少运算量，提高运算速度。

5. IMDCT(改进离散余弦反变换)

解码时反量化结束后会得到信号数据，逆向离散余弦变换 IMDCT 就是利用本身的计算公式，对这些数据信号进行变换的。这是个非常庞大的过程，不仅运算量大而且算法很复杂，因此此过程的优化算法也是本论文讨论的重点内容。

6. 频率反转和子带合成

编码时为提高 MDCT 的效率，对相应数据做出了频率反转的处理，所以要正确地得到解码结果,恢复出原来的信号，就要有频率反转的过程，补偿编码时进行的频率反转[10]。频率反转是对上面逆向离散余弦变换过程产生的的 32 个子带进行标号,把奇数号子带中的奇数号数据进行反相操作。

频率反转后的信号是 32 个子带的频域信号，子带滤波的作用实质上就是要把这频域信号转变为时域信号，恢复出原始的 PCM 信号。IMDCT 变换后会产生 32 个频带，子带综合滤波器会把这 32 个频带生成 64 个中间值，这些中间值先缓存在 FIFO 缓存器中转成 1024 个数值。下一步把这些数据按一定的规律输出，输出值构成了 512 个矢量值，并与窗函数经过相乘叠加生成 32 个 PCM 输出值。

§2-2 MP3 音频格式

MP3 码流的最小组成单位是帧，编码过程中每个帧进行数据压缩时，都要按 MPEG Layer3 音频标准规定的帧格式压缩，帧格式如图 2.2 所示。解码时也要按照 MPEG Layer3 音频标准进行解码，一帧包含 1152 个采样值。MP3 码流由多个音频帧组成，每个帧主要组成部分为：帧头(frame header)，错误校验码（CRC），边带信息（side_info），主数据（main data）以及附加数据[11]。其中 CRC 和附加数据不是必须的，边信息包含解码所需的参数。在解码过程中，码流通过帧头里的信息确认数据的开始，把帧头里的参量同步字和状态信息提取出来，就可以恢复出 1152 个 PCM 的采样值。主数据中有两个粒度 granule 0 和 granule 1，这两个粒度分别有 576 个 PCM 采样值。

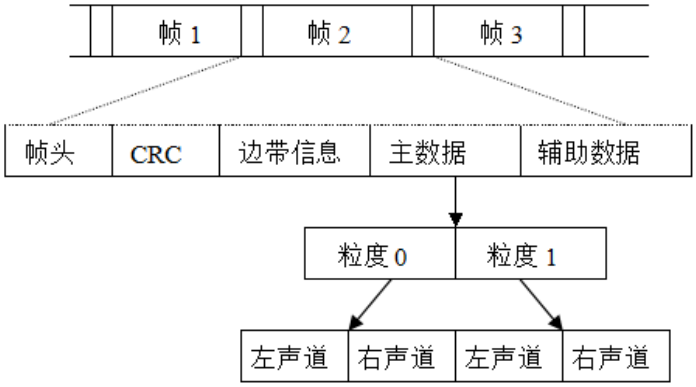

图 2.2 帧格式示意图
Fig.2.2 Schematic diagram of frame format

如图 2.2 所示是双声道模式下的示意图，包含左右两个声道。主数据分成两个粒度，每个粒度都含有左右两个声道的信息，因此可以说主数据中，实际上含有粒度 0 和粒度 1 各自的左右声道的比例因子和 Huffman 数据[12]。

2-2-1 帧头格式

对于固定频率的 MP3 文件，所有帧的帧头格式都是相同的，帧头格式如图 2.3 所示，MP3 帧头共有 32 位，包含的信息有同步字、采样率等信息。

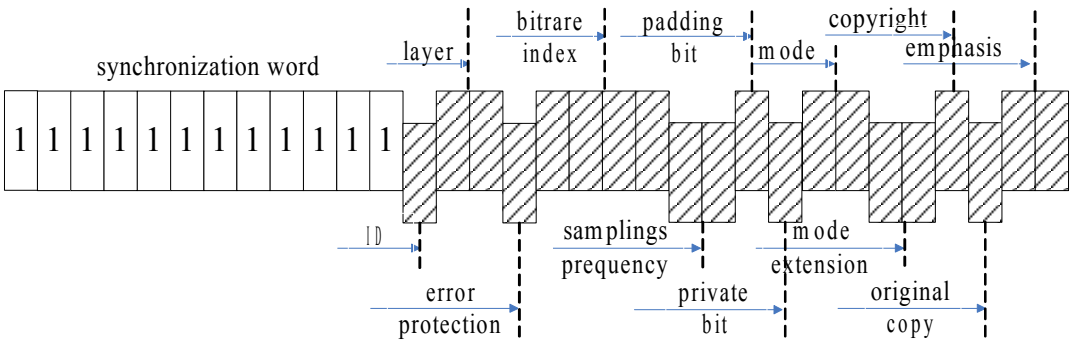

图 2.3 帧头信息格式图
Fig.2.3 Chart of frame header format

帧头字节说明如表 2.1 所示：

表 2.1 帧头字节长度及使用说明
Table 2.1 length and directions of MP3 frame head

名称	长度	说明
同步信息（syncword）	12	标记起始位，所有位均为 1，即“FFF”。
标识码（ID）	1	以 1 个比特“1”表示为 MPEG-1 Audio。
编码层（layer）	2	00-保留；01-Layer3;10- Layer2,11- Layer1。
保护位（protection_bit）	1	0-校验，1-不校验。
比特率索引（bit_index）	4	表明该帧每秒钟的数据量大小，单位 Kbps。
采样频率（sampling_frequency）	2	00-44.1kHz,01-48kHz,10-32kHz,11-保留。
填充位（padding_bit）	1	频率 44.1kHz 时，1-有填充位，0-无填充位。
私有位（private_bit）	1	私有使用位。
模式（mode）	2	00-立体声，01-联合立体声，10-双声道，11-单声道。
扩展模式（mode_extension）	2	详见表 2.2。
版权（copyright）	1	0-数据流没有版权，1-受版权保护。
原件/拷贝（original/copy）	1	0-拷贝，1-原始。
加重位（emphasis）	2	使用的解码加重类型。

扩展模式当声道模式为01时使用，使用说明如下。

表 2.2 MP3 扩展模式索引
Table 2.2 Index of MP3 mode extension

索引值	中边立体声	深度立体声
00	关	关
01	关	开
10	开	关
11	开	开

2-2-2 CRC 错误校验

错误校验字 CRC 是可选的，用来检测数据传输错误，它是 16 位的奇偶校验字。一般 MP3 编码算法中不常用 CRC 校验。

2-2-3 帧边信息

每帧 MP3 数据中，帧头之后是边带信息，包含解码解码所需的参数如表 2.3 所示。对单声道下，边带信息需占用 136 位，而双声道模式，需要占用 256 位。大致共包含四个方面的信息，指针（main_data_begin）、公用的边信息、粒度 0（granule 0）的左右声道边信息和粒度 1（granule 1）的左右声道边信息[13]。

表 2.3 边信息结构
Table 2.3 Side_info structure

声道	Main_data_begin	Private_bits	Scfsi	Side_info_gr0	Side_info_gr1
双声道	9	5	4	59	59
单声道	9	3	8	118	118

主数据开始位(main_data_begin)、保留位(Private_bits)和比例因子选择信息(Scfsi)这三个变量是公用边带信息，是两个粒度的左/右声道所共用的三个变量。

main_data_begin: 是一个向前偏移量，表示主数据开始的位置，当本帧的主数据开始位置在前面的帧中，时就要由 main_data_begin 的值判断，向前面的帧中偏移多少个地址单元。

Scfsi:比例因子选择位，作用是确定粒度 0 和粒度 1 是否公用比例因子。当两个粒度的比例因子相同时，就可以把 Scfsi 位置为 1，需要比例因子时只需要发送粒度 0 的值，可以节省空间。

Private_bits：保留位，留作私用，可自定义其中的内容。

各粒度各声道下的边带信息格式相同，内部参量说明如下所示:

表 2.4 边信息变量长度
Table 2.4 Variable length of Side_info

Part2_3_length	Big_values	Global_gain	Scalefac_compress	Window_switching_flag
(12，24)	(9，18)	(8，16)	(4，8)	(1,2)

Table_select	Granule0_count	Granule1_count
(l5，30)	(4，8)	(3，6)

Block_type	Mixed_block_flag	Table_select	Subblock_gain
(2,4)	(l，2)	(l0，20)	(9，18)

Preflag	Scalefac_scale	Count1table_select
(1,2)	(1,2)	(1,2)

其中括号内表示的分别是单声道和双声道时所所需要的比特数。

内部参量说明如下：

Part2_3_length:表明主数据中 scalefactor 和 Huffman 码字的长度。

Big_values：可以提供 Huffman 解码的大值区的码字个数。

Global_gain：全局缩放因子，反量化步骤中每个 PCM 值都要乘以全局缩放因子。

Scalefac_compress：用来检索比例因子的长度。

Block_type：0 表示长块, 1 表示开始块, 3 表示结束块, 2 表示短块。

Window_switching_flag：表示是否出现特殊块，0-没有出现特殊块，1-出现了特殊块。

Mixed_block_flag：混合窗标识，1 表示低频的 3 个频带用到了长块，其余频率子带用到了特

殊块。

Table_select：指定不同区域的使用的霍夫曼码表的参量。

Subblock_gain：解码时反量化过程中短块数据所用的偏移量。

Granule0_count：霍夫曼码字大值区第 1 个子区域的频带个数。

Granule1_count：霍夫曼码字大值区第 2 个子区域的频带个数。

Preflag 和 Scalefac_scale：反量化解码时所需要的变量。

Count1table_select：小值区霍夫曼的码表选择的变量。

2-2-4 主数据

主数据包含比例因子 scalefactor 和霍夫曼码字，每帧主数据包含粒度 0 和粒度 1，主数据个结构如图 2.4 所示：

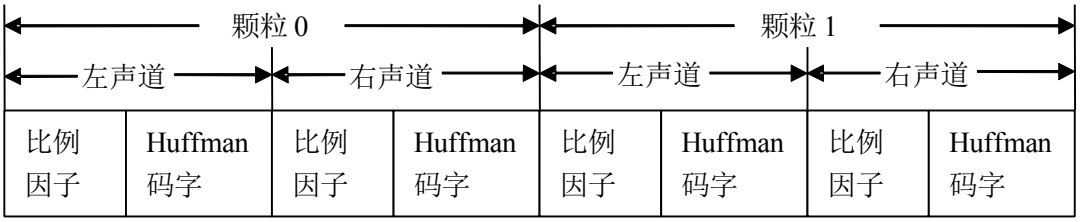

图 2.4 主数据的格式示意图

Fig.2.4 schematic diagram of Main_data format

主数据从码流中分离出来后，会存储在主数据 RAM 单元中，供下面的比例因子和 Huffman 模块访问。因为在缓存区中的数据采用的比特池技术，因此主数据有可能有一部分存放在之前的帧中，解码时要根据 main_data_begin(主数据开始)来确定本帧主数据开始地址单元，main_data_begin 是一个偏移量指出主数据开始向前偏移的地址单元数，它是独立的不会受到其他参数的影响[14]。

2-2-5 辅助数据

在 MP3 码流中辅助数据是可选的，可以包括作者信息和歌名，是与 MP3 解码本身没有关系的数据，通常是自定义信息。

§2-3 比例因子

通过 Huffman 解码可得到 576 个频率值值，每个值代表一个频率线。然后要进行反量化处理，反量化的过程是以若干条频率线为单位进行的，而不是一以条频率线为单位进行的，这些频率线组成的频带称为比例因子频带[15]，反量化时比例因子频带内的频率线公用一个缩放因子。不同的块类型其比例因子的个数、频带的划分和含义都不同。

（1）长块类型

长块类型的数据（block_type=0,1,3）划分成 21 个比例因子频带，每个频带内的频率线共用一个比例因子，共有 21 个比例因子。在 44.1kHz 的采样率下，比例因子频带的划分如表 2.5 所示。频率线 0 至 3 属于比例因子频带 0，带宽为 4，即此频带内有 4 个数据；频率线 30 至 35 属于比例因子频带 7，带宽为 6，频带内有 6 个数据。频率线 418 至 575 未被归属于某个比例因子频带，因为属于这个频带的频率线在进行反量化时，系统提供默认的反量化因子。

下表 2.5 表述的是当采样频率是 44.1KHz 时，长块比例因子频带的划分方式。

表 2.5 比例因子频带划分
Table 2.5 Division of scalefactor band

比例因子频带	频带宽度	起始索引	结束索引
0	4	0	3
1	4	4	7
2	4	8	11
3	4	12	15
4	4	16	19
5	4	20	23
6	6	24	29
7	6	30	35
8	8	36	43
9	8	44	51
10	10	52	61
11	12	62	73
12	16	74	89
13	20	90	109
14	24	110	133
15	28	134	161
16	34	162	195
17	42	196	237
18	50	238	287
19	54	288	341
20	76	342	417

比例因子被编码于主数据中，解码时需要知道每个比例因子的比特长度，根据该长度从主数据中取出特定位宽的数据，即可得到对应的比例因子。不同频带内的比例因子长度不同，解码时要根据比例因子的长度取出一定长度的数据，进一步确定频率线的比例因子。要确定比因子的长度可以由 scalefac-compress 搜索 slen 索引表得到，比例因子的长度索引如下所示。

表 2.6 比例因子长度索引表
Table 2.6 Index of scale factor length

scalefac-compress	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Slen1	0	0	0	0	3	1	1	1	2	2	2	3	3	3	4	4
Slen2	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

对于长块类型的数据，划分成了 21 个频带，第 0-10 个频带，比例因子的长度是相等的，可以通过检索 slen1 得到长度值；第 11-20 个频带的比例因子长度同样是相等的，可以检索 slen2 得到长度值。长块数据类型的比例因子的总长度为 $part2\_length=11\times slen1+10\times slen2$。

（2）短块类型

短块类型(block_type=2)的 576 个值代表 192 条频率线，每条频率线上面有三个值，对应不同的窗类型，

每个窗对应不同的比例因子。这 192 条频率线分为 12 个比例因子频带，所以共有 36 个不同的比例因子[16]。下表 2.7 表述的是当采样频率是 44.1KHz 时，短块比例因子频带的划分方式。

表 2.7 短块类型比例因子频带划分
Table 2.7 Division of short blocks scalefactor bands

比例因子频带	带宽	起始索引	结束索引
0	4	0	3
1	4	4	7
2	4	8	11
3	4	12	15
4	6	16	21
5	8	22	29
6	10	30	39
7	12	40	51
8	14	52	65
9	18	66	83
10	22	84	105
11	30	106	13

频率线 0 至 3 属于比例因子频带 0，带宽为 4，每条频率线上有 3 个值，因此频带 0 内实际有 4×3=12 个频域值。频率线 136 至 192 没有归属于某个比例因子频带，因为属于这些频率线在进行反量化时，系统提供默认的反量化因子。

比例因子的长度可以通过参量 scaledac_compress 搜索 slen 检索表得到。第 0-5 个比例因子频带，每个频带有 3 个比例因子，6 条频带对应 18 个比例因子，这 6 个频带的比例因子长度是一样的，可以搜索 slenl 得到；第 6-11 个频带的长度可以搜索 slen2 得到。短块数据类型的比例因子总长度为 $part2\_length = 6\times 3\times slen1 + 6\times 3\times slen2$。

（3）混合块类型

混合块（block_type=2 且 mixed_block_flag=1）解出来的 576 个值按先后顺序分为长块和短块两部分，第一部分(前 36 个值)是长块部分，此 36 条频率线参照表格 2.5 划分为 8 个比例因子频带；第二部分(后 540 个值)是短块部分，包含 180 个频率线，此 180 个频率线参考表格 2.7 划分为 9 个比例因子频带，属于短块的比例因子频带 3 至频带 11,每个频率线上有 3 个值，代表 3 个比例因子[17]。因此混合块共有 8+9×3=35 个比例因子。

长块的比例因子长度可以搜索 slenl 得到，同样短块第 3-5 个频带的比例因子长度搜索 slenl 得到，而剩余的第 6-11 个频带的长度可检索 slen2 的值得到。由此可得混合块类型中比例因子总长度为 $part2\_length = (8+3\times 3)\times slen1 + 6\times 3\times slen2$。

§2-4 Huffman 解码

解码是编码的逆过程，即是根据码字查寻 Huffman 码表，还原出原始数值。MP3 帧结构中的主数据包括比例因子和 Huffman 码字，把由边带信息参量提取出来后，通过计算可以得到比例因子的总长度

part2_length，再利用简单的减法运算可计算出，霍夫曼码字的总长度=part2_3_length - part2_length。Huffman 解码后可以恢复出的 576 个频率线，它们分为三个区域分别是大值区、小值区和零值区，这三个区域按频率从高到低排列，如图 2.5 所示。

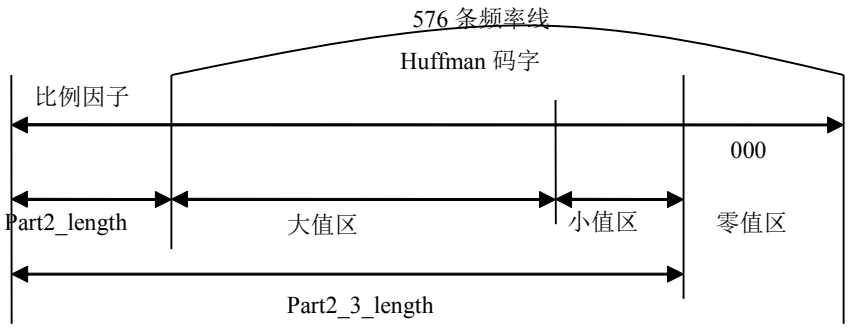

图 2.5 Huffman 码字结构示意图
Fig.2.5 Structure diagram of Huffman code word

区域边界由边带信息指定，只有大值区和小值区的频率线会出现在 MP3 码流中，零值区的频率线值全为 0，因此不用编码也不必出现在码流中，解码时只需对零值区的频率线补 0，直至共得到 576 个解码值。

大值区：此区域的每个 Huffman 码字对 2 个频率线 x 和 y 进行编码，参量 big_value 表示大值区 Huffman 码字的总个数，因此大值区可解码得到 2*big_values 个频率线。被编码的频率线值小于等于 15，如果超过 15 则只对 15 进行编码，而超出的部分被表示成扩展值(ESCAPE)，放在 Huffman 码字之后,扩展位宽是 linbits。被编码的数据为该值的绝对值，符号位单独传送，对于非 0 频率线，需要设置 1 bit 的符号位。

大值区的 32 个 Huffman 码表有一定的特点，其中，使用前 16 个 Huffman 码表进行编码的主数据纯粹由 Huffman 码字组成，不包含扩展值，且表 0 为零值表，表 4 和表 14 未使用。后 16 个 Huffman 码表均包含扩展值，表明 Huffman 码字后面可能带有扩展值。表 17 至表 23 这七个表，与表 16 除了扩展值的位宽不同，码表内容完全相同，表 25 至表 31 共七个表与表 24 也是只有位宽 linbits 不同。因此，Huffman 解码时只需处理前 13 个非 0 值表、表 16 以及表 24 共 15 个 Huffman 码表即可，其它码表以此为基础通过变换即可得到。

小值区：小值区使用 2 个特殊的 Huffman 码表，具体的码表选择信息由参量 count1table_select 提供。此区域的一个 Huffman 码字对 4 个频率线 v、w、x、y 进行编码[18]，且每个频率线只有 3 种可能值-1、0 和 1。被编码数据非 0 时，符号位单独传送。小值区边带信息中没有直接表示小值区码字个数的参量，但当 Huffman 解码共使用了 part2_3_length 长度的主数据后，即表明完成了小值区的解码。

零值区：这个区域的频率线不会在码流中出现，因为此区域频率线的值全为 0，解码时只要补零即可。

Huffman 解码得到的频域值按比例因子频带从低到高排列。短块类型时，每个比例因子频带内部还按照按窗(window)的顺序排列，每个窗内的频率线由低到高排列。Huffman 解码完成后，把处理过的数据通过总线存储到 RAM 中，以供反量化的模块使用。

§2-5 反量化

反量化的过程就是利用反量化的公式、边信息等进行计算以恢复出 576 个频率线真实值的过程，反量化时不同的快类型选用的公式是不同的。令 Huffman 解码后的值为 is，反量化得到的值为 xr，反量化根据

不同的块类型，选用不同的公式[19]：

长块数据类型的公反量化式如（2.1）：

$$xr_i = sign(is_i)*|is_i|^{\frac{4}{3}} * \frac{2^{\frac{1}{4}(global\_gain[gr][ch]-210)}}{2^{(scalefac\_multiplier*(scalefac\_l[gr][ch][sfb]+preflag[gr][ch]*pretab[sfb]))}} \tag{2.1}$$

短块数据类型的公反量化式如（2.2）：

$$xr_i = sign(is_i)*|is_i|^{\frac{4}{3}} * \frac{2^{\frac{1}{4}(global\_gain[gr][ch]-210-8*subblock\_gain[gr][ch][sfb][win])}}{2^{(scalefac\_multiplier*(scalefac\_s[gr][ch][sfb][win]))}} \tag{2.2}$$

在反量化公式中，频谱值 is 首先增大到原来的 4/3 次幂，之后再乘以符号位 $sign(is_i)$。$global\_gain$ 变量是整个声道的全局量化步长。系统常数 210 是量化步长，另一个作用是避免编码过程中出现全“1”的情况而影响同步字。在编码器中比例因子 $scalefac\_scale$ 采用对数量化的形式，步长为 2 或 $2^{1/2}$，若 $scalefac\_scale$=0 则 $scalefac\_multiplier = 0.5$，否则 $scalefac\_multiplier = 1$。$preftab$ 和 $preflag$ 只在长窗中有效，$preflag = 1$ 表示采用高频预加重，$pretab[sfb]$ 的作用是查表得出每个频率子带的预加重值。$scalefase\_s$ 表示短窗的比例因子，$scalefacse\_l$ 表示长窗的比例因子。$subblock\_gain$ 表示短窗运算时子带中更细化时的变量。变量 win_switch_flag 和 $blocktype$ 共同作用确定是否重排序。

§2-6 重排列

重排列是对短块数据进行重新排列处理的，长块数据的顺序按频率由低到高排列。因为在编码的时候，MDCT 变换后，短块类型的数据按照频带、窗、频率的顺序排列，为了提高编码的效率，对他的排列顺序进行了调整，变为频带、频率、窗。而解码就是编码的反过程，为了正确地解码，在反变换后必须要把短块数据的排列顺序恢复成频带、窗、频率[20]。重排列的过程完全不会影响频率线的归属性，频率线的顺序也不会改变。确定是否重排列的参数是变量 window_switching_flag 和 block_type。

§2-7 立体声处理

MPEG LayerIII支持常用的简单的双声道模式和立体声模式，这两种情况下不需要进行数据处理。此外还支持联合立体声模式 MS 立体声和增强立体声，MS 立体声是无损的立体声模式，选用什么模式是由参量 $mode$ 和 $mode\_extension$ 决定的，MS 立体声要求两个声道的数据重建[21]。

MS 立体声模式下，码流传递是中间/旁边声道值，这种模式下只需传送一个均值和一个小值，如公式 4-2 所示，M 的值表示均值，而 S 的值非常非常小，选择 MS 立体声模式可以减少传输的数据长度[22]。

编码公式如下 2.3 所示，L 代表左声道，R 代表右声道，M 代表中间声道，S 代表旁边声道。

$$M = \frac{L+R}{\sqrt{2}}, S = \frac{L-R}{\sqrt{2}} \tag{2.3}$$

MS 立体声模式下的解码过程中，得到左右声道值 L/R 的公式为:

$$L = \frac{M+S}{\sqrt{2}}, R = \frac{M-S}{\sqrt{2}} \tag{2.4}$$

强度立体声模式中左声道传送的是指定的幅值，而右声道传送的是立体声的位置 is_pos。解码时利用

式(2.5)、(2.6)解出左右声道信号。

$$is\_ratio = \tan(is\_pos \times \frac{\pi}{12}) \tag{2.5}$$

$$L = L \times \frac{is\_ratio}{1+is\_ratio}, R = L \times \frac{1}{1+is\_ratio} \tag{2.6}$$

当立体声处理的过程在左右声道处理完大值区和小值区进入零值区时，因为零值区两个声道的频率值等于 0，就可以省去此后计算的步骤，直接得出结果 0。

§2-8 混叠重建

MP3 编码过程中 MDCT 处理之后，产生 32 个频带，相邻的频带之间会产生相互影响，为了尽可能低地降低这种影响，编码时就做了减少混叠的处理，所以解码时要完全恢复处理之前的信号。而混叠是针对长块类型的数据的，所以解码时除了重建长块类型的数据，还包括了混合块中的前两个子带的部分，两个子带做一次蝶形运算[23]。

长块类型数据有 32 个子带，混叠重建时每个相邻子带之间要进行 8 次蝶形运算。运算方法是前一个子带的后 8 个和下个子带的前 8 个进行运算，所以第一个子带的前 8 个和最后一个子带的后 8 个频率值，不需要参与运算，这样就需要 31 组这样的运算[24]。蝶形计算公式如式 2.7 所示，每次蝶形计算的输入记为 xu、xd，输出记为 xu'、xd'，其中的 cs 和 ca 参数由 MP3 音频标准给出。

$$xu' = xu \times cs_i - xd \times ca_i, \qquad xd' = xd \times cs_i + xu \times ca_i \tag{2.7}$$

由于零值区特性在混叠重建进行到零值区时，可以直接添加结果 0，减少运算量。蝶形运算的过程示意图如图 2.6 所示。

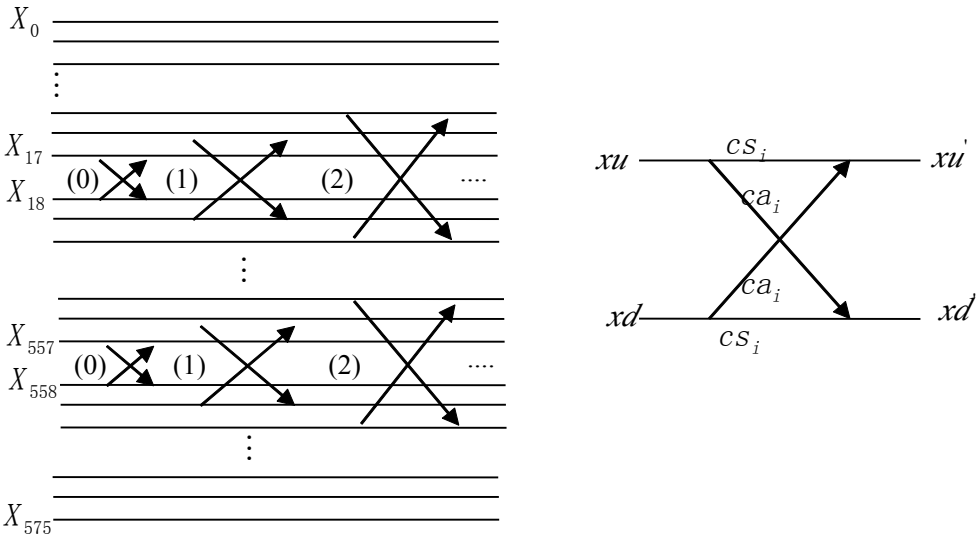

图 2.6 混合重建处理

Fig.2.6 Alias Reconstruction processing

§2-9 IMDCT 变换

IMDCT 是以子带作为单位来进行计算的，解码过程大致包括三部分 IMDCT 变换、加窗和叠加，下面进行解析。

（1） IMDCT 变换

X_k 表示输入值，x_i 表示输出值，运算公式如式 2.8 所示：

$$x_i = \sum_{k=0}^{n/2-1} X_k \cos\left(\frac{\pi}{2n}(2i+1+\frac{n}{2})(2k+1)\right) \quad \text{其中 i = 0,1 ,… n-1} \qquad (2.8)$$

长块类型的数据时 n=36，短块类型的数据时，n=12。

长块类型 IMDCT 变换时，每个子带有 18 条频率线，输入也就是 18 个值，会输出 36 个结果，计算过程是输入值与系数相乘之后再进行 17 次累加。总共 36 行，每行 18 个变换系数，可能取值有 36×18=648 种[25]。由于余弦函数的对称性，所以基于各行的系数有对称性，只需要计算 18 种变换结果，其他数据可根据以下关系得出。

$$x_{17-i} = -x_i,\ i = 0,1,2\cdots,8$$
$$x_{53-i} = x_i,\quad i = 18,19,20,\cdots,26$$

（2） 加窗处理

将输出根据不同的块类型，选择不同的窗函数加窗：

- 长块类型数据

block_type=0(长块)

$$z_i = x_i \sin\left(\frac{\pi}{36}(i+\frac{1}{2})\right), 0 \le i \le 35 \qquad (2.9)$$

block_type=1(长块起始)

$$z_i = \begin{cases} x_i \sin\left(\frac{\pi}{36}(i+\frac{1}{2})\right), & 0 \le i < 18 \\ x_i & 18 \le i < 24 \\ x_i \sin\left(\frac{\pi}{12}(i-18+\frac{1}{2})\right), & 24 \le i < 30 \\ 0 & 30 \le i < 36 \end{cases} \qquad (2.10)$$

block_type=3(长块结束)

$$z_i = \begin{cases} 0 & 0 \le i < 6 \\ x_i \sin\left(\frac{\pi}{12}(i-6+\frac{1}{2})\right), & 6 \le i < 12 \\ x_i & 12 \le i < 18 \\ x_i \sin\left(\frac{\pi}{36}(i+\frac{1}{2})\right), & 18 \le i < 36 \end{cases} \qquad (2.11)$$

三种长块类型中，IMDCT 变换结果 x_i 直接乘以窗函数 w_i，可得到 36 个加窗结果 $z_i = x_i * w_i$。

- 短块类型数据 block_type=2(短块)

短块的窗函数为：

$$w_i = \sin(\frac{\pi}{12}(i+\frac{1}{2})), \quad 0 \le i < 12$$

短块加窗如下：

$$y_i^{(k)} = x_i^{(k)} w_i \quad 0 \le i < 12, 0 \le k < 3$$

加窗后的结果必须重叠连续，加窗结果为：

$$z_i = \begin{cases} 0 & 0 \le i < 6 \\ y_{i-6}^{(1)} & 6 \le i < 12 \\ y_{i-6}^{(1)} + y_{i-12}^{(2)} & 12 \le i < 18 \\ y_{i-12}^{(2)} + y_{i-18}^{(3)} & 18 \le i < 24 \\ y_{i-18}^{(3)} & 24 \le i < 30 \\ 0 & 30 \le i < 36 \end{cases} \tag{2.12}$$

短块类型中，每次 IMDCT 变换的结果乘以窗函数w_i，得到 12 个加窗结果y_i。3 次加窗结果之间要进行部分叠加和补 0，才能得到最终的 36 个结果z_i。

（3） 叠加运算

叠加的计算过程是， 当前子带的 36 个z_i值分成前 18 个值和后 18 个值两部分，前 18 个值跟前一个子带的z_i的后 18 个值相加，当前子带的后 18 个值保存起来，跟下一个子带的z_i的前 18 个值相加[26]，运算公式如式（2.13）所示：

$$\begin{aligned} &\text{result}_i = z_i + s_i, \quad 0 \le i \le 17 \\ &\text{s}_i = z_{i+18} \quad 0 \le i \le 17 \end{aligned} \tag{2.13}$$

IMDCT 变换过后就不在区分长块和短块，变换后的 576 个值由低到高总共分成 32 个子带，1 个子带包含 18 个值。

§2-10 频率反转和子带综合滤波

频率反转是在 IMDCT 之后，对 IMDCT 的子带输出值进行反向处理，即频率反转补偿，来校正多相滤波器组的频率反转[27]。方法是对 32 个输出的子带有低到高标号为 0 到 31，将奇数号子带的奇数采样值乘以-1。

频率反转处理完之后是子带综合滤波的过程，下面是具体操作方法。

1. 从标号 0 到 31 的 32 个子带中，分别提取出 32 个值，这些值经过公式（2.14）变换后得到 64 个中间值[28]。

$$x_i = \sum_{k=0}^{31} x_k \cos(\frac{\pi}{64}(16+i)(2k+1)) \quad 0 \le i \le 63 \tag{2.14}$$

2. 把上个步骤 1 得到的 64 个中间值，放入 FIFO 缓存区。FIFO 是先进先出的缓存设备，初始化为 0，长度是 1024。

3. 从骤 2 的 FOFO 缓存区中的 1024 个值，按特定顺序提取一半的值，提取完后组成一个含有 512 个值的矢量 U。

4. 进行加窗处理，向量 W=Ui*Di，Di 是窗函数系数，是 MP3 官方协议中给定的值。因此矢量 W 同样包含了 512 个值。

5. 矢量 W 中包含的 512 个值中，连续分成 16 组，每组 32 个值，顺次提取每组 32 个值中的一个值，把每次能取得 16 个值相加，最后就会得到 32 个时域信号[29]。

第三章 MP3 解码器的结构设计

§3-1 设计目标

本设计是基于 Altera 公司 Cyclone II 系列的 EP2C35F672C6 的 FPGA 开发平台进行 MP3 解码器的设计，最终解码得到的 PCM 值用 FPGA 开发板实现声音的播放。Altera 公司 2004 年推出了新款 Cyclone II 系列 FPGA 器件，包含了许多新的特性，在电信、计算机外部设备、工业以致于汽车市场上取得了不错的成绩。利用 Cyclone II 软件，可以完成包括仿真在内的 FPGA 开发过程。

对于 MP3 音频解码器的设计，按照自顶向下的设计思路，在对解码算法进行研究与分析的基础之上，首先从系统级设计入手，进行顶层结构设计和功能模块划分，确定整体硬件结构后，分别对每个模块进行 RTL 级设计和功能仿真，最后再完成整体硬件系统的集成与验证。

本设计的设计目标，需要满足以下要求：

1、解码满足实时性的要求；

2、支持双声道模式；

3、采用 Verilog HDL 语言进行设计。

§3-2 硬件结构设计

为了系统设计的方便，对系统进行了模块的划分，分成了不同功能的子模块，不同子模块之间的连接方式有两种点对点和总线。一般情况下硬件实现时选用总线的方式，总线是各个模块公用的，采用的是时分复用方式，这种方式既节约了电路的面积，又不会在同一时间产生访问冲突。采用总线的方式连接时，要求各个子模块有统一的标准的接口[30]。

采用总线方式时，系统设计中的设备有主从之分，本质上来说主设备是发送方，从设备是接收方，比如存储设备就是从设备。当主设备需要对从设备进行读/写操作时，就会发出请求信号，从设备接收到信号后，就做好读/写的准备。一个系统中可以存在多个主设备，它们对从设备的访问就要划分优先权，这时，就要有负责区分优先权的设备，称作仲裁器。当多个主设备同时发送请求信号时，由仲裁器决定哪个主设备有使用权限[31]。如图 3.1 所示，主设备是模块 1 和模块 2，从设备是存储器。当模块 1 需要把数据写入存储器时，就先给仲裁器发送写请求，仲裁器接收到信号后，就会释放总线给模块 1，这时模块 1 就可以把要写入的数据通过总线存储在储存器中。

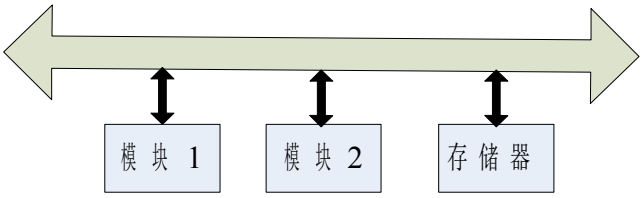

图 3.1 基于总线的连接方式
Fig. 3.1 Bbased on bus connection

鉴于MP3解码的实时性要求，为加快解码速度，在系统架构上采用左右声道并行解码的方式。并行解码可以保持左右声道的解码同步性，便于解码流程的控制。虽然MP3解码器有实时性要求，但实时处理并不完全意味着高速，只要解码速度达到实时处理的要求，再考虑速度优化就没有多少意义，此时可着重考虑面积优化。

为了声音可以连续流畅地播放出来，MP3解码器就必须满足实时解码，高效快速地进行解码，本设计采用左右声道并行同步解码，但在两个声道内部采用的是串行解码的方式[32]。左右声道的解码是相对独立进行的，解码的时候流程更容易控制；左右并行解码已经能达到实时性的要求了，内部采用串行解码是为了节约存储空间，减少电路设计的面积，合理利用资源。各个模块共享总线和RAM单元，因此可以选用六选一MUX选择器作为仲裁器，六个模块的工作状态可以作为MUX的6位的输入控制信号。

MP3解码器的结构如图3.2所示：

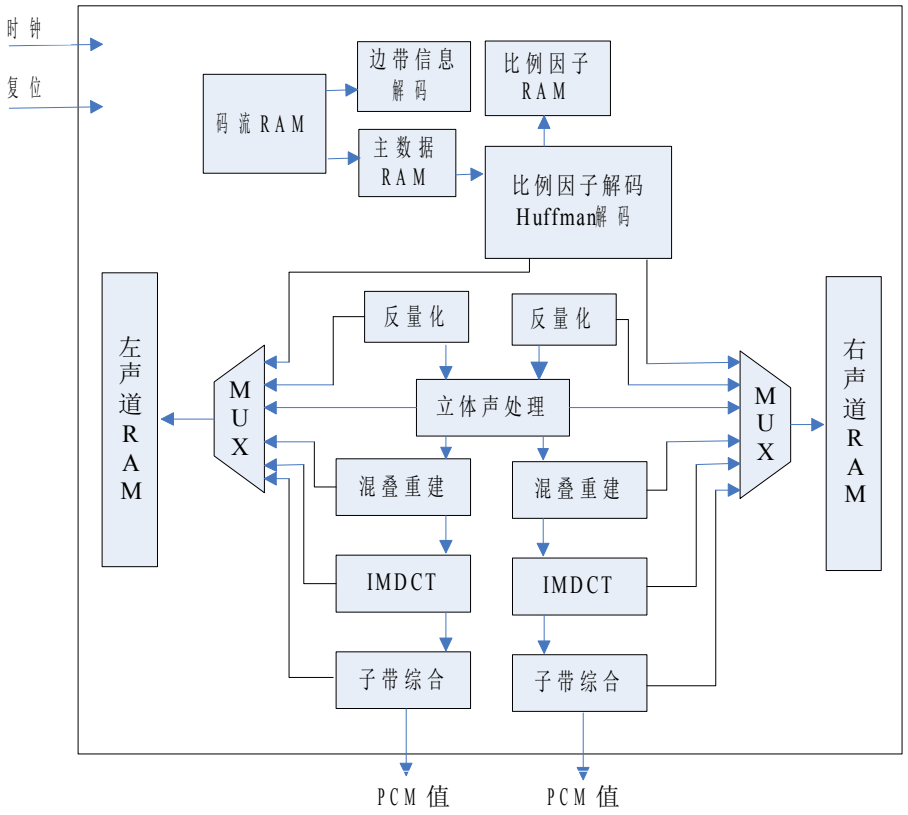

图 3.2 MP3 解码器的结构图

Fig.3.2 Chart of MP3 decoder structure

MP3 解码过程大致分码流解析部分和数值计算部分。

（1）码流解析部分主要包括边带信息解析、比例因子解码和霍夫曼解码。MP3 码流中包含解码所需的参量如同步字等，通过参量解析模块把同步字提取出来，就开始了一帧的解码，同时把得到的主数据暂存在主数据 RAM 单元中，这个 RAM 实际作用就是作为比特池缓存使用的；然后比例因子模块就可以访问主数据 RAM 单元得到比例因子的值，把提取到的比例因子也存放在 RAM 单元；Huffman 解码子模块访问主数据 RAM 后，可以解码得到左声道的 576 个频域值[33]，通过数据总线存储到左声道的 RAM 中，然后再次读取主数据 RAM，解码右声道的信息，同样把得到得到的右声道频域值通过总线存储到右声道的 RAM 中。

（2）码流解析部分完成后就开始了数值分析的部分，由于左右声道并行解码，左右声道的解码原理是相同的，因此下面我们以左声道为例进行分析说明。

如上图 3.2 所示，针对这种多主设备的总线架构，其仲裁机制比较简单，因为每个模块顺序启动，任意时刻只有一个子模块在工作，不会出现同时有多个主设备访问总线的冲突现象，因此只要某个模块处于工作状态，便可将总线资源分配给该模块。

在码流分析的部分，Huffman 解码模块已经把解析出的左声道的 576 个值存储到了左声道 RAM 中，之后这个模块会把总线的使用权限交给反量化模块。反量化模块开始工作后，通过总线访问左声道 RAM 的数据，并依次读取出来，利用反量化的公式对数据进行处理[34]，处理完后的数据再通过总线存入左声道 RAM 单元中，反量化模块处理完毕。

接下来反量化模块释放总线的权限给立体声子模块，从图 3.2 可以看出，这个模块同时连接了两个总线，可以同时处理左右声道 RAM 中的数据。立体声模块得到总线使用权限后，通过总线读取左声道 RAM 中的数据，把处理过的数据再次存入左声道 RAM 单元，释放总线权限。剩下的混叠重建和 IMDCT 对左声道 RAM 的处理方法是同样的原理，再次不再叙述。最后子带综合滤波模块得到总线使用权限时，首先读取左声道 RAM 中的数据，并进行子带综合滤波处理，计算完毕后得到的数据就不再次存入左声道 RAM 中了，而是通过播放控制模块直接输出[35]。

在左声道的工作过程中，各个模块要频繁的访问左声道 RAM 单元，为了节省解码时间，相邻的模块间上个模块读取数据处理完毕后，也可以把数据结果直接交给下个模块，就像反量化模块和重排列模块，完全可以结合在一起[36]。因为重排列过程不需要对数据进行计算，只要把反量化后的数据进行排列即可，反量化模块工作时，把得到的数据反量化处理后，可以直接传递给重排列模块，把经过重排后的数据再存储进左声道 RAM 单元，这就省去了反量化模块的存入和重排列的读取过程，两个过程减少了一半对 RAM 单元的访问量。

第四章 MP3 解码器各模块的设计

§4-1 参量解析模块

参量解析模块检索到码流中的同步字后，确定一帧解码的开始。MP3 码流中的各个参量按固定格式紧密排列，参量之间没有明确的界限，本解析模块只能根据每个参量的比特长度按顺序进行数据分离，即参量解析过程串行进行。此外，必须保证每次解析过程的正确性，否则一个参量的取数过程出现错误(多取或少取)，后续的码流解析步骤都会出错。

MP3 音频文件暂存于 FPGA 器件的片内 RAM，进入解码器的码流位宽为 32bits，而参数解析模块的每次取数长度不定，因此需要设置内部缓存器，将输入的码流值寄存起来，以便每次从中获取不同长度的数值。

为加快解码速度，实现在最少的时钟周期内将数据解析出来，本文设计中使用桶型移位器进行数据移位以实现码流的分离，通过 QuartusⅡ软件的 Mega Wizard 管理器可以生成桶型移位器的 IP 核，将其设置为通过组合逻辑实现任意移位，并立刻得到移位结果。

取数模块设置两个 32bits 的寄存器 D0、Dl 用来寄存码流值，并作为 64bits 桶型移位器的输入。取数时，首先根据之前总的取数长度对桶型移位器进行左移位，把已经解析过的数据移出，保证桶型移位器的高位输出端始终是未解析的数据。取桶型移位器的高 32 位输出作为取数模块的数据输出 Dout，控制模块得到数值 Dout 后根据参量的比特长度从中分离出所包含的参量值，因此取数长度 Get_numb 不能超过 32 位，将其位宽设置为 5bits。每次输入的取数长度 Get_numb 逐次累加，寄存后累加结果的低 5 位 Shift_numb 作为移位控制量，控制着桶型移位器的左移个数。

§4-2 比特池缓存区模块

MP3 音频标准对比特池的空间大小作了规定，作为比特池最少要有 960 个字节。本设计根据数据的长度将作为缓存区的主数据 RAM 单元设计成 2048 个字节，主数据 RAM 有两个端口，分别对应两个独立的总线读写总线和地址总线[37]。读写总线由边带解析模块进行控制，帧解码开始后，把得到的主数据存入主数据 RAM 单元中。在存储数据的过程中，每个地址写入一个数据，写完一个数据后的地址会自动加一，直到地址从 0 增加到 2047 时，如果继续写入数据，地址就会自动变为 0，数据会存入地址 0 的空间，并且原来的数据被格式化[38]。帧格式中的主数据不可能超过 2048，所以不会出现格式化上一帧没有解码的数据的情况。

由于采用了比特池技术，本帧的数据可能存储的别的帧中，所以需要指针 Frame_start 对每帧的开始位置进行控制，它表示本帧主数据开始的地址[39]。示意图如图 4.1 所示，Main_data_begin 中的数据表示本帧的数据在其他帧中存入的主数据的长度 Main_data_end 中的数据表表示帧的数据的解码在缓存区中的结束位置。

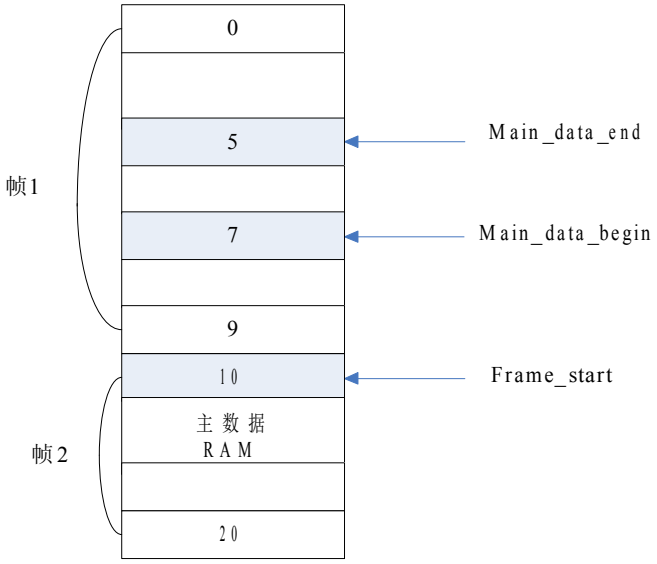

图 4.1 比特池控制示意图
Fig.4.1 Schematic diagram of bit pool control

图 4.1 的示意图的含义是，假设有帧 1 和帧 2 两个帧结构，帧 1 数据存储地址是 0-9 十个单元，帧 2 的存储地址是 10-20 十个单元，帧 2 作为正在解码的数据帧。此时指针 Frame_start 的值就是 10，即指向本帧的开始地址，当 Main_data_begin 的值是 3 时，就表示帧 2 的主数据有部分存储在帧 1 中，存储的字节长度为 3，那么帧 2 主数据的开始位置就是从本帧开始的位置向上的第 3 个单元，也就是从 7 开始。Main_data_end 的值等于 5 时，表示帧 1 的数据解码在单元 5 时结束。假如 Main_data_end 的值等于 8，Main_data_begin 的值是 3，这时帧 1 的主数据解码到 8 时才结束，而 Main_data_begin 的值是 3 表示帧 2 中，有三个值存储在帧 1 中，说明帧 2 的主数据存在错误，解码时就会直接开始帧 3，放弃对帧 2 的解码。

§4-3 比例因子和霍夫曼解码模块

4-3-1 比例因子模块

帧解码开始后，参量模块先启动，把分解出的比例因此存放到比例因子 RAM 中，把主数据存放到主数据 RAM 中[40]。然后比例因子模块开始工作，访问数据 RAM 单元以得到比例因子，结束后 Huffman 模块开始访问主数据 RAM，把得到的左/右声道的频率值存放到左/右声道 RAM。图 4.2 所示的是比例因子模块的流程图。

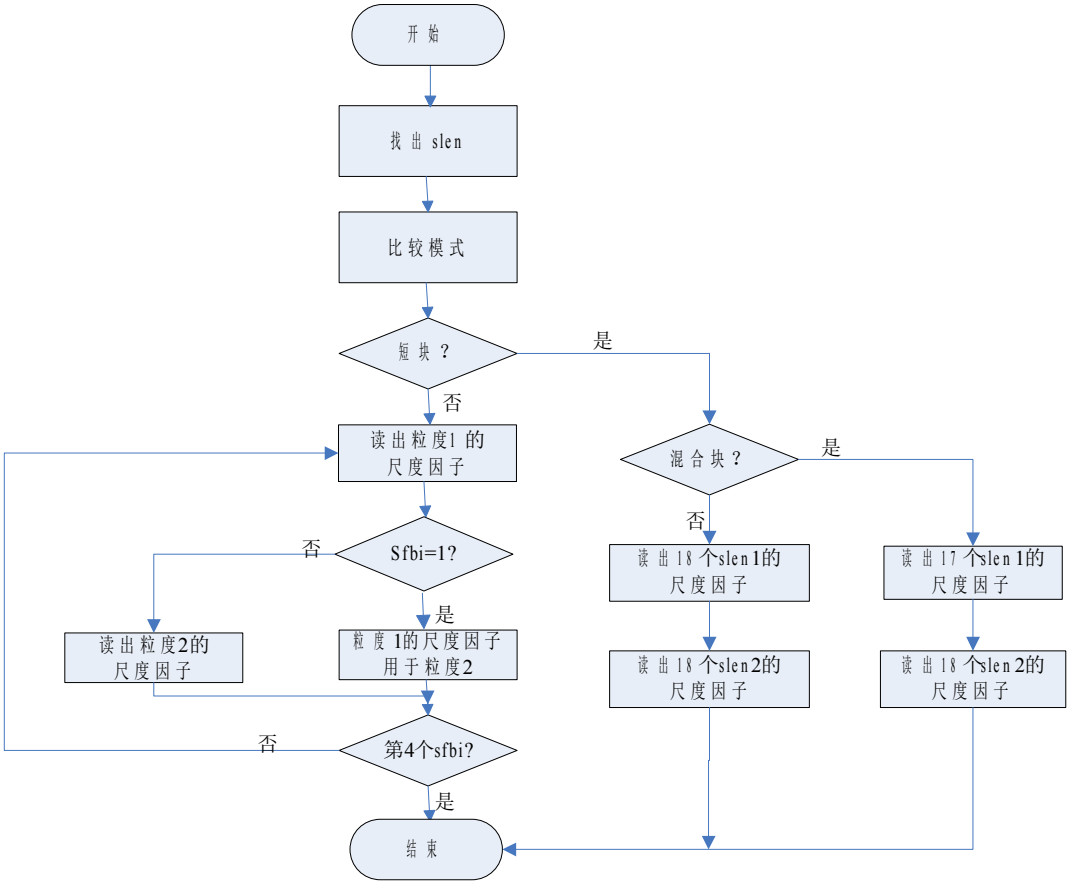

图 4.2 比例因子模块流程图

Fig.4.2 Flow chart diagram of scale factor module

流程开始后首先要找出 slen 判断块类型。长块类型时，有 21 个频带，前 11 个频带的比例因子长度可以检索 slen1[41]。频带 11 到 20 的比例因子长度可以检索 slen2，从主数据中获取 10 次 slen2 长度的比例因子写入存储单元。为便于计算，把未纳入比例因子频带的频域值归属到频带 21，作为最后一个比例因子频带，对应的比例因子值为 0。对应比例因子长度是 part2_length=11*slen1+10*slen2。当数据是短块类型时，比例因子总长度为 part2_length=6*3*slen1+6*3*slen2，因此需要读取 18 次 slen1 长度，18 次 slen2 长度的数据存入 RAM 单元。混合块时，比例因子总长度为 part2_length=(8+3*3)*slen1+6*3*slen2 要读取 17 个 slen1 长度,18 次 slen2 长度的数据存入 RAM 单元中。

其主程序如下所示：

```
always_comb  begin
        unique case(block_type)//判断块类型
            LONG_BLK :      begin   max_subband = 20; slen1_limit = 10; end
            SHORT_BLK:      begin   max_subband = 35; slen1_limit = 17; end
            MIXED_BLK:      begin   max_subband = 34; slen1_limit = 16; end
            default:        begin   max_subband = 20; slen1_limit = 10;   end
        endcase
    end
always_comb begin
```

```
        part2_length = 0;//默认值
        unique case (block_type)
            LONG_BLK   :    part2_length = 11*slen1+10*slen2 ;
            SHORT_BLK :     part2_length = 18*(slen1+slen2);
            MIXED_BLK :     part2_length = 17*slen1 + 18*slen2;
            default    : ;
        endcase
    end
```

4-3-2 霍夫曼解码模块

Huffman 解码主要是对压缩的数据进行解码，首先访问主数据 RAM 单元，通过解码得到比例因子，存放到比例因子 RAM 中。同时从同步信息中得到解码所需要的参量，根据的得到的参量确定霍夫曼解码所需要的信息，再查找 RAM 中建好的所需要的霍夫曼码表，然后就可以对频率线进行解码[42]，输出 575 条频率线的频域量化值，解码流程图如 4.3 所示。零值区的值可以不用处理，直接补零就可以得到解码值。直到 576 条频率线都完成后，把解码数据传输到共享的声道 RAM 中，供反量化模块使用[43]。

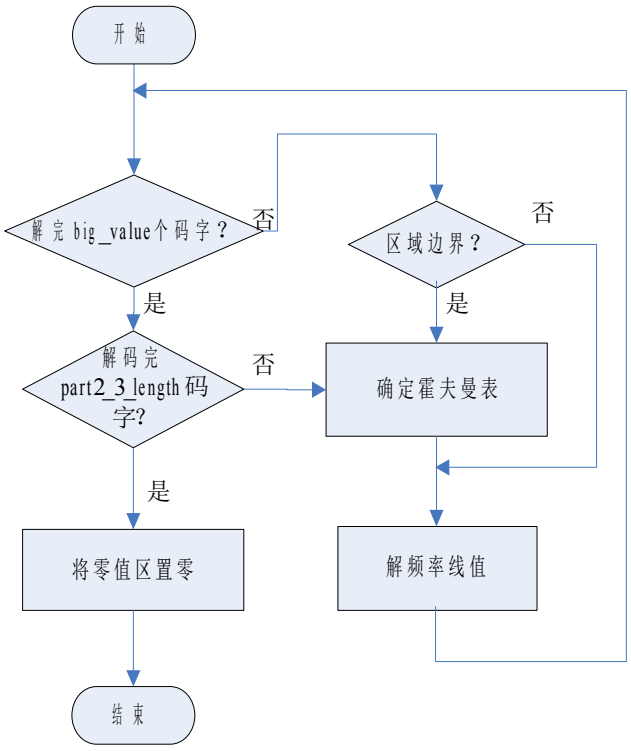

图 4.3 Huffman 解码流程图

Fig.4.3 Flow chart of Huffman decoding

般大值区占用的比特数较多，为了进一步提高大值区 Huffinan 码编码效率，将其细分为 Region0，Regionl，Region2 三个区域，允许每个区域选择不同的码表，而三个区域的边界也不是固定的，在编码端根据数据块特性查表决定。码表选择的主程序为：

```
begin
  if(start)
```

```
        remainBits = part3length;
    if(bread)
        remainBits = (remainBits - bnbits);
end
wire r1 = region0 < caddr;
wire r2 = region1 < caddr;
wire [4:0] select;
always_comb
begin
    case({r2,r1})
        2'b11: select     <= table_select[4:0];          // in region 2
        2'b01: select     <= table_select[9:5];          // in region 1
        2'b00: select     <= table_select[14:10];        // in region 0
        default: select <= table_select[4:0];
    endcase
end
```

Huffllllan 解码过程逐步进行，每次解码得到两个或四个频率线值保存在公共 RAM 单元，大值区和小值区解码结束后，到达零值区边界，此时可直接向公共 RAM 单元写入数值 0，直至共得到 576 个频域值。本模块需要记录零值区边界值，并作为输出信号供后续计算模块使用，以减少数据计算量。

§4-4 反量化和重排列模块

反量化过程是根据(式 2-1)和(式 2-2)的反量化公式进行计算的，由于要实现 $x_i^{\frac{4}{3}}$ 的计算需要非常复杂的电路。所以就要对计算方法进行优化，我们可以采用除 8 查表法，只需要基本查表法的约 8 分之一的存储空间[44]，只需要把 0 到 1023 的 4/3 次方预先计算好保存在 ROM 单元。如果频率线索引值不大于 1023，反量化时就可以直接查到这个表格，但是这种方法的缺点是需要 256K 的存储空间。为了快速得到结果，又节省空间，继续优化查表法，方法如下，可以节约 32K 的存储空间。

(l)如果 $x_i < 1024$ 时，可以从表中直接查找到反量化结果。

(2)如果 $x_i \geq 1024$ 时，先计算 $x_i / 8$，用除 8 后的结果查找表中的数据，最后的得到的值再乘 16。由式(4.1)可以来验证这种方法的正确性[45]。

$$\left|x_i\right|^{\frac{4}{3}} = 16\left|\frac{x_i}{8}\right|^{\frac{4}{3}} \qquad \text{（式 4.1）}$$

表格的建立过程如下所示：

打开 Quartus II 软件点击 file/new/memory file/memory initialization file 建立初始化储存器

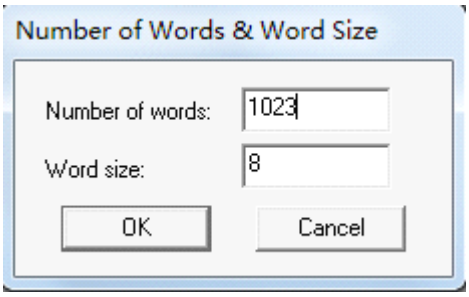

图 4.4 初始化存储器的建立

Fig.4.4 Create memory initialization

点击确定后，生成.hex 类型的文件，里面的初始化数值为 0。把 0 到 1023 的 4/3 次方存放在相应的单元中。如图 4.5 所示

Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	00	40	28	22	32	22	2B	35
008	3F	25	2B	30	36	3D	21	24
010	28	2B	2F	32	36	39	3D	20
018	22	24	26	28	2A	2C	2E	30
020	32	34	37	39	3B	3D	3F	21
028	22	23	24	25	26	28	29	2A
030	2B	2C	2E	2F	30	31	33	34
038	35	36	38	39	3A	3C	3D	3E
040	3F	20	21	22	22	23	24	24
048	25	26	26	27	28	28	29	2A
050	2B	2B	2C	2D	2D	2E	2F	30
058	30	31	32	33	33	34	35	36
060	36	37	38	39	3A	3A	3B	3C
068	3D	3D	3E	3F	20	20	20	21
070	21	22	22	22	23	23	24	24
078	24	25	25	26	26	27	27	27

图 4.5 hex 文件生成图

Fig.4.5 hex file generation diagram

硬件计算单元由四级流水线实现，四个功能块分别为：读取频域值；通过查表计算频域值的 4/3 次方；进行相乘和移位计算；写回频域值。时序图如图 4.6 所示，第一级流水线中地址寄存器连续增加，可连续读取三个频域值，第二级流水线中查表操作连续工作三次，第三级流水线中乘法操作亦是连续工作三次，第四级流水线把得到的三个处理结果依次写回公共 RAM 单元。这样每处理三个数据，只需六个时钟周期，因此处理总的 576 条频率线只需要 576×2 个时钟周期。

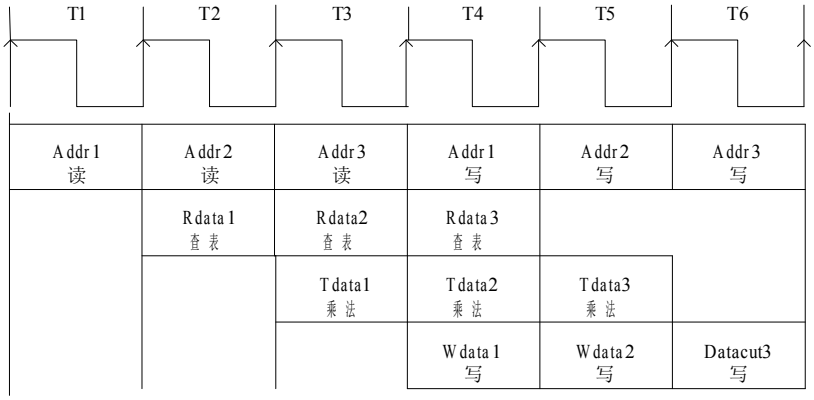

图 4.6 反量化模块工作时序图
Fig. 4.6 Sequence diagram of requantizer module

上面设计核心是生成访问公共 RAM 的地址，读数据时根据三个读地址连续读入三个数据，得到计算结果后，需要连续产生三个写地址，以便把处理结果写回原 RAM 单元。如图 4.6 所示，在使用地址 Addrl 准备读取输入数据后的第四个时钟周期，可以把处理结果写入地址 Addrl 所指向的存储单元。因此把读地址值寄存三个时钟周期，即可作为对应的写地址值。

短块类型的数据进行反量化的过程是以频带为单位的，当频带内的数据被读取出来时，依次对读取的数据进行处理。在硬件结构设计时，把反量化模块和重排列模块融合在一起，所以经过反量化处理后的数据，不会直接传递给左右声道 RAM，而是传送给重排列模块重新排列顺序[46]。然后重排后的数据通过总线传送给 RAM 单元中，而且存放数据时，要存储在新的存储空间 RAM2 中，以免把还没有进行反量化的数据覆盖掉。RAM2 的长度必须大于最长的频带的带宽。

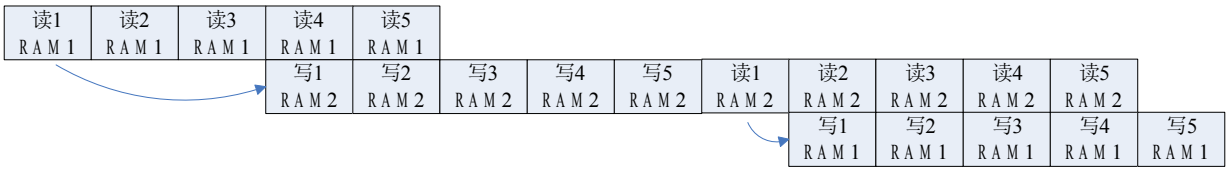

图 4.7 反量化和重排列的访存示意图
Fig.4.7 schematic diagram of inverse quantization and relining

如图 4.7 所示，在从公共 RAM 单元(即 RAM1)中读取输入数据的第四个时钟周期，即可以把计算结果按照重排后的地址暂存于缓冲区 RAM2。一个比例因子频带内的数据全部处理完毕后，均保存在 RAM2 缓冲区，而且缓冲区中暂存的是重排后的计算结果。然后即可顺序读出 RAM2 中的数据写回公共 RAM 单元。

向 RAM1 单元写入第一个数据的同时，可以准备从 RAM2 中读取第二个数据，这样下个时钟周期可以继续向 RAM1 中写入第二个数据，同时准备从 RAM2 中读取第三个数据，依次类推。因此采用流水线设计处理一个比例因子频带时，可以实现连续地读取输入数据，中间只需要等待四个时钟周期，即可连续地写回处理结果。

为了加快解码速度，对反量化和重排列模块进行了融合，也就是说反量化的数据直接传输给重排列模块，以减少对 RAM 的访问量。在写入暂存区 RAM2 之前，重排列模块已经对数据进行了重排列的处理，重排列流程如下图 4.8 所示。

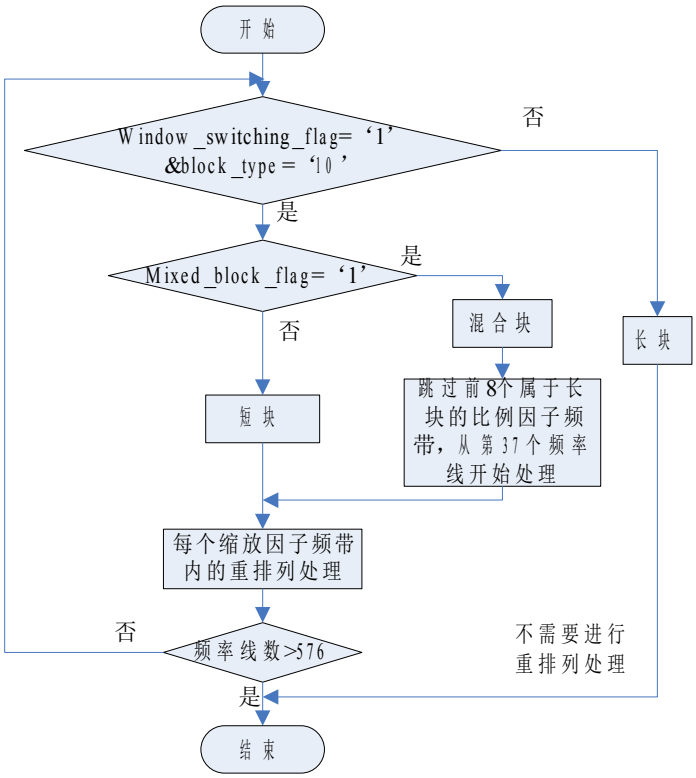

图 4.8 重排列处理流程图
Fig. 4.8 Flow chart of reorder processing

反量化之后短块的数据排列顺序是按照频带、频率线、窗的，重排序是按频带、窗、频率线的顺序重新排列，即每个缩放因子频带 3 个窗，每个窗 6 个频率线采样值。现将图 4.8 中的一个频带内的重排序过程描述如下。

一个频带内按照频率线、窗的顺序排列的数据存储在数组 a[频率线号][窗号]中，其中频率线号为 0 到 5，窗号为 0 到 2，这样，共有 18 个频率线采样值，记为 1-18，即

a[0][0…2]={ 1,2,3 }, a[1][0…2]={ 4,5,6 }, a[2][0…2]={ 7,8,9 }

a[3][0…2]={ 10,11,12 }, a[4][0…2]={ 13,14,15 }, a[5][0…2]={ 16,17,18 }

按照窗、频率线的顺序排列后的结果 b[窗号][频率线号]为:

b[0][0…5]={ 1,4,7,10,13,16 }, b[1][0…5]={ 2,5,8,11,14,17 },

b[2][0…5]= { 3,6,9,12,15,18 }

这样就完成了一个频带内的重排序处理，同时将重排序之后的结果存储在重排序模块的主存储区内，方便下个模块调用。在处理完 576 条频率线之后，重排序模块将结束标志 finish 置‘1’，完成本模块的工作。

§4-5 混叠重建模块

混叠重建模块是对长块类型的数据进行处理的，对反量化的数据，在相邻子带间进行蝶形运算，按蝶形运算的算法，将一对频率线进行 4 次乘运算，一次加和一次减运算后，得到一对新的频率线量化值，最后把这些新值写入声道 RAM 单元。长块类型数据有 32 个子带，混叠重建时每个相邻子带之间要进行 8 次蝶形运算。运算方法是前一个子带的后 8 个和下个子带的前 8 个进行运算，所以第一个子带的前 8 个和

最后一个子带的后 8 个频率值，不需要参与运算，这样就需要 31 组这样的运算[48]。

蝶形计算的程序如下：

```
MUL1:           begin
                    temp1 <= temp;
                end

MUL2:       begin
                    temp2 <= temp;
                    check <= 1;
                end

MUL3:           begin
                    temp3 <= temp;
                end
MUL4:           begin
                    temp4 <= temp;
                    check <= 0;
                end

PREPARE_WRITE:begin
                        address    <= cnt;
                        write_data      <= temp1 - temp4;
                    end

WRITE: begin
                address <= distance;
                write_data <= temp2 + temp3;
            end
```

在处理到零值区时，可以用计数器记下零值区的边界值，充分利用零值区的特性，当计算到边界值时，以后就可以不用在计算，直接得出结果零，当处理完所有的子带后，模块结束工作。之前所有的模块都是基于频域进行的，下面的过程则是频域数据向时域转换的过程。

§4-6 IMDCT 模块

混叠重建后的数据，存储到公用的 RAM 单元中，下一步就要启动 IMDCT 模块。IMDCT 处理模块分成三个部分 imdct_trans 模块、windowing 模块、mul 模块。

imdct_trans 模块先以子带为单位进行 IMDCT 变换，其中用到乘法的时候，调用 mul 模块进行处理，mul 模块输出的积返回给 imdct_trans 模块进行处理；同时，为了满足 IMDCT 累加运输的要求，将积存储到内部存储区进行备份。因为 MP3 的每个粒度中的 576 条频率线从低到高分为 32 个子带，每个子带包含 18 条频率线，因此，imdet_trans 模块将以每个子带内的 18 条频率线值为输入，产生 36 个多相滤波器的子带采样值。最后输出 32×36 个子带采样值。这 32×36 个子带采样值最后都存储到内部存储区内，方便后面模块调用。

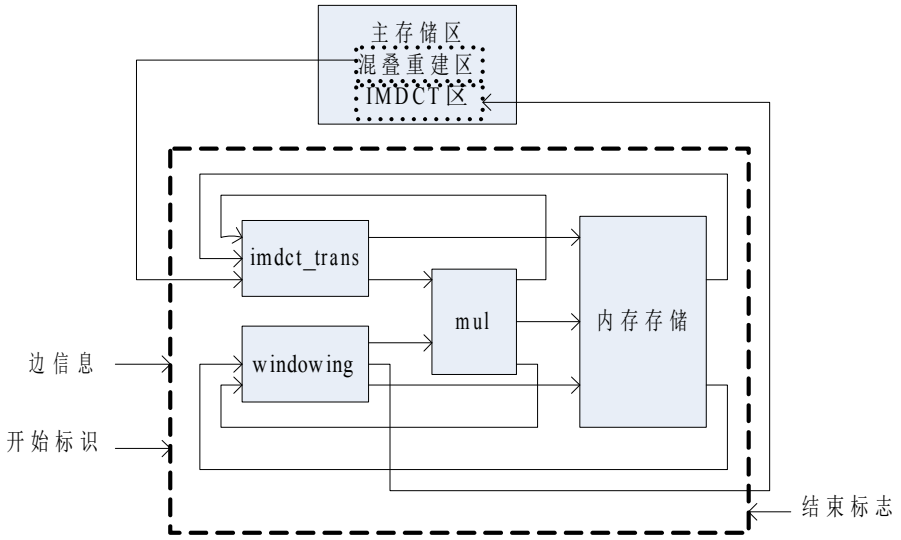

图 4.6 IMDCT 模块数据流图
Fig.4.6 Data flow diagram of IMDCT module

图 4.6 为 IMDCT 处理模块的数据流图。在 imdct--trans 模块处理完之后，windowing 模块开始工作。其将内部存储中 32×36 个经过 IMDCT 变换的数据逐一读出，并根据边信息提供的 block_type 信息，以子带为单位依照式(2.9)-(2.12)，进行加窗运算。其中用到乘法的地方，也调用 mul 模块进行处理，其调用过程同 imdct_trans 模块。最后进行叠加处理，依照式(2.13)，将每一个子带变换出来的值的前半部分与前一个子带的后半部分相加，并把后半部分保留与下一个子带的前半部分相加。windowing 模块输出的结果为 32×18 个子带采样值，将这些数据存储到主存储区的 IMDCT 区之后，结束本模块的工作。

IMDCT 变换得到的 36 个变换结果，需要保存到缓冲区 RAM；加窗操作时，从缓冲区中读出变换结果，乘以窗系数后再写回；叠加运算时，再次访问缓冲区 RAM，读出乘窗结果进行加法操作，得到计算结果后写入公共 RAM 单元。如此需要设置缓冲区以保存中间结果，而且需要反复地访问缓冲区 RAM，因此考虑将三个步骤融合在一起进行硬件实现，当 IMDCT 变换得到一个中间结果时，紧接着便进行加窗乘法和叠加操作，得到最终处理结果后再写回公共 RAM，这样即减少了频繁的访存次数，又节省了存放中间结果的缓冲区 RAM 单元。

IMDCT 变换是以频带为单位进行的从公共 RAM 单元读取一个子带的频域值，与对应的窗函数系数相乘，逐个累加后得到一个 IMDCT 变换结果。由快速算法可知一个乘累加值实际表示两个中间结果，分别对这两个中间结果乘窗、叠加，即可得到两个最终的计算值。如此循环 18 次，便得到 36 个计算值，其中前 18 个数据作为最终输出写回公共 RAM 单元，后 18 个数据保存起来供下个颗粒使用。32 个子带全部参与计算后，共得到到 576 个数据。把这些值存储的公用的 RAM 单元中，供子带合成滤波模块使用。

§4-7 子带合成滤波模块

把经过频率反转后的 32×18 个值从公用 RAM 中提取出来，32 个频带分别从 0 到 31 进行标号。从 32×18 个值中，每隔 18 个提取一个，总共可以抽取到 32 个值[50]。流程图如 4.7 所示，这 32 个值经过公式（2.14）变换后，得到 64 个中间值。这 64 个中间值存放进类似先进先出的 FIFO 缓存器中，数据存放的长度是 1024。再把 FOFO 缓存区中的 1024 个值，提取一半的值后组成一个含有 512 个值的矢量 U。对这 512 个值进行加窗处理，也就是用窗函数的系数 D 乘以 U，加窗后得到另一个有 512 个值的矢量 W。把向量 W 连续分

成 16 组，每组 32 个值，顺次提取每组 32 个值中的一个值，把每次能取得 16 个值相加，最后就会得到 32 个时域信号。这 32 个信号就是最后的是与输出 PCM 值。

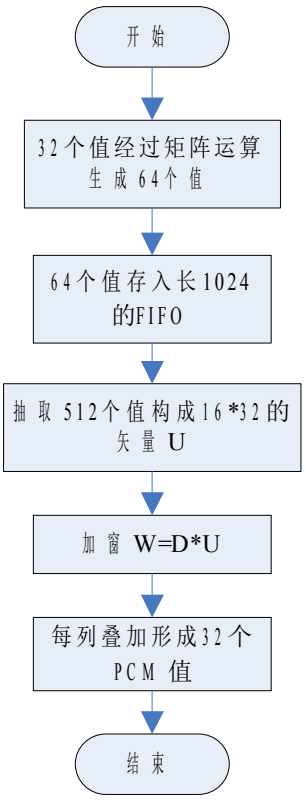

图 4.7 子带合成滤波器处理流程图

Fig4.7 Subband synthesis filterband flow chart

从以上步骤中可知，一个粒度要进行 18 次这样的操作，才能处理完粒度 0 的 576 个值，把处理完的值存储在子带综合滤波器内，然后返回继续对粒度 1 进行解码；如果粒度 1 的解码完成后，则返回寻找同步字，对下一帧进行解码。

第五章 系统仿真及分析

§5-1 功能仿真

5-1-1 模块 IP 核建立过程

完成 MP3 音频解码器的设计时，使用 Quartus Ⅱ软件编写程序并进行功能仿真，以 Cyclone Ⅱ系列的 EP2C35F672C6 的 FPGA 开发板做为开发平台，Quartus Ⅱ软件是现阶段非常流行的是仿真工具，具有强大的仿真功能和友好的图形化界面[51]。

(1) 用 Verilog HDL 语言进行编程，如图 5.1 是读写 MP3 文件的程序编程界面。

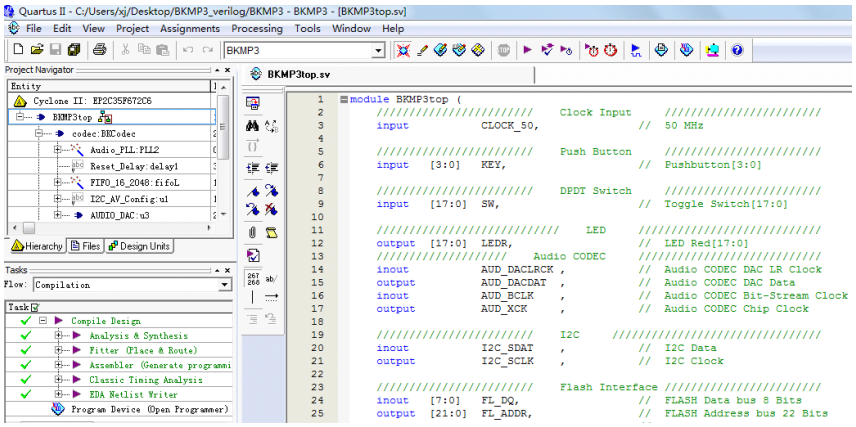

图 5.1 Quartus Ⅱ 软件的编程界面

Fig.5.1 Programming interface of Quartus Ⅱ software

（2）本文采用模块化处理，先进行模块例化，程序编写完成后进行编译，点击 Processing▶ Start Compilation 开始编译，完成后会出现成功提示，如图 5.2 所示。如果不成功则需要继续调试，直至编译成功。

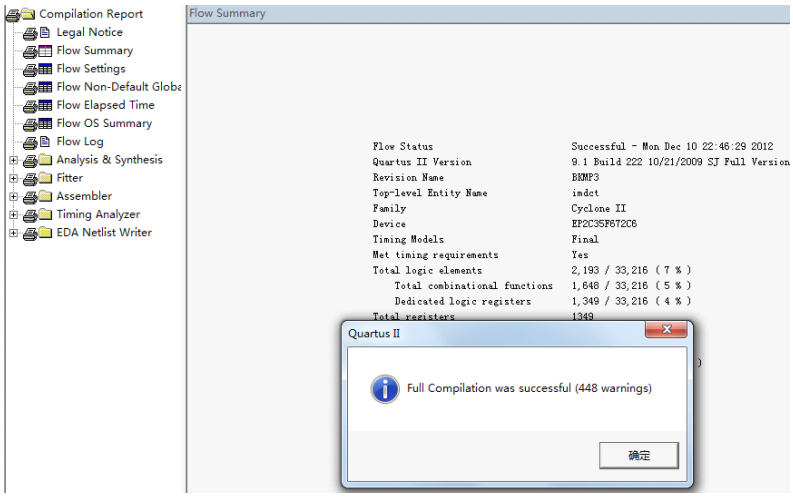

图 5.2 编译成功界面

Figure 5.2 Interface of compiler success

（3）主数据缓存器 MainFiFo

在 QuartusII 软件中，在选择 MegaWizard Plug-In Manager(自带宏模块) 工具生成 altera IP 核并编译仿真。由于各个模块的配置过程相似，我们以 MainFiFo 为例，对模块进行参数配置如图 5.3 所示，本设计的输入数据 8 位，FIFO 大小是 2048 个字。

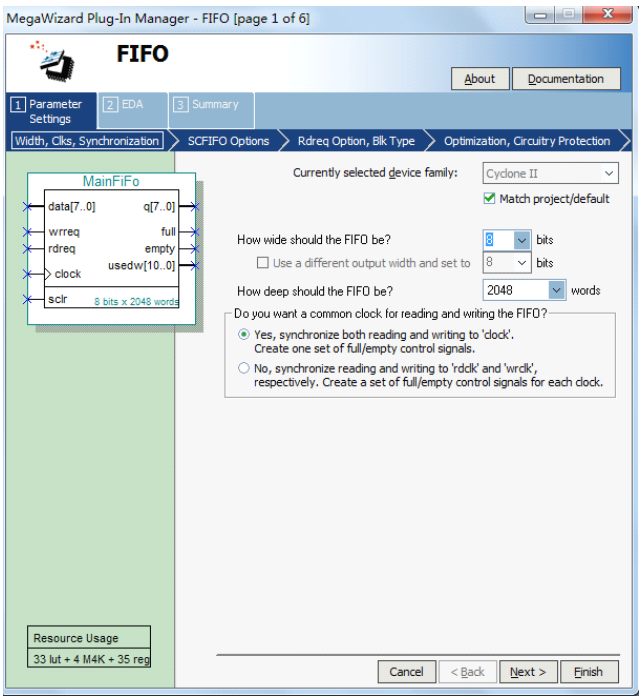

图 5.3 MainFiFo IP 核的创建

Fig.5.3 Create MainFiFo IP core

为加快解码速度，实现在最少的时钟周期内将数据解析出来，本文设计中使用桶型移位器进行数据移位以实现码流的分离，通过 Quartus II 软件的 Mega Wizard 管理器可以生成桶型移位器的 IP 核，将其设置为通过组合逻辑实现任意移位，并立刻得到移位结果。

5-1-2 主要模块端口及仿真

（1）MainFiFo 模块

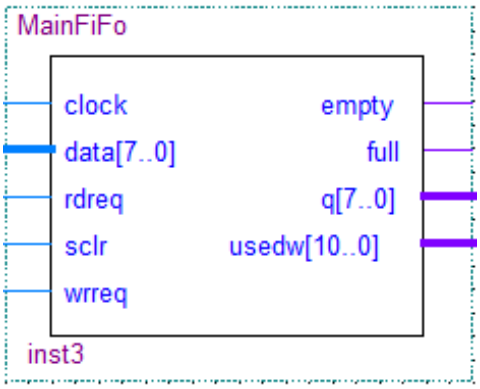

图 5.4 MainFIFO 模块图

Fig.5.4 Chart of MainFIFO module

读写操作都需要通过时钟来控制，单端口 RAM 只有一个读写口，同时只能做读或者写操作，有独立的地址总线和数据总线，数据总线又分为读数据线和写数据线，其主要的端口说明如下：

Clock:时钟信号。

Address:地址线，为读写操作所共用。

Wdata:写操作对应的数据线，传递写入 RAM 的数据。

Wren:写使能信号，当写信号 Wren 为高电平时是写操作，将写数据线上的值 Wdata 写入地址 address 所指向的内存空间。

Rdata:读操作对应的数据线，当写使能信号 Wren 为低电平时是读操作，将地址 address 所指向的内存空间值读出到读数据线上。

Full：FIFO 已满时输出信号。FIFO 满时输出高电平，就不在写入新的数据。

Cyclone Ⅱ RAM 的输入地址要经过一级固有的寄存器，仿真波形图如图 5.5 所示，存储单元内的数据在地址有效后的下个时钟周期被读出；写操作时，写数据在写控制信号和写地址的同步下，于当前时钟周期被写入存储器[52]。因此 MP3 解码子模块与总线之间的数据传输接口简洁统一，有访问公共存储器的地址线、读数据线、写数据线、写控制信号。

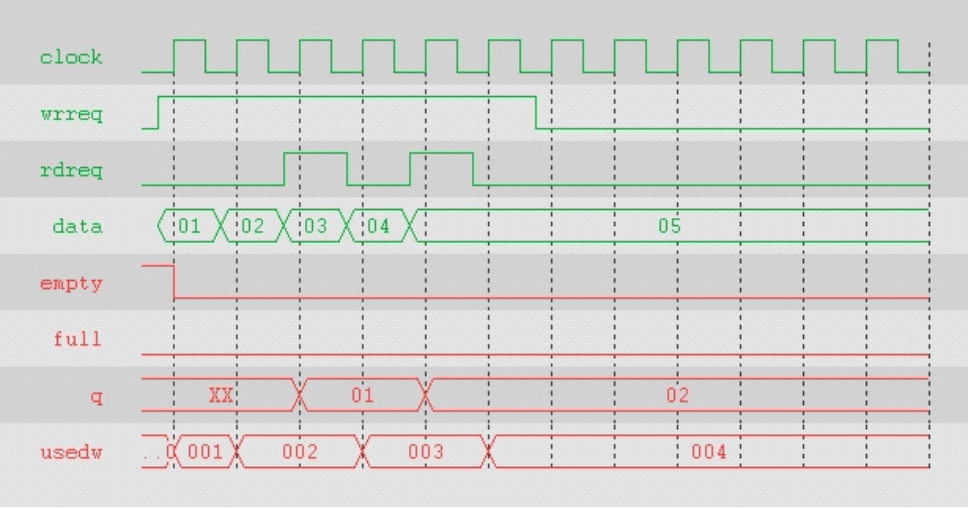

图 5.5 FIFO 的读写操作

Fig.5.5 Chart of FIFO read and write operation

（2）Huffman 解码模块

主要的端口说明：

table_select:大值区的码表选择。

count1_table_select:小值区的码表选择。

din_bit：比特池中读出的数据。

valid_bit：比特池有效标志位。

region0，region1，region1：大值区内划分的三个子区域。

part3length：比例因子和 Huffman 码字的总长度。

read_bit：表示正在读比特池中的数据。

nbits_bit：比特池中数据的比特数。

write_common_ram:写数据标志位。

addr_common_ram：要写入内存的地址。

dout_common_ram：输出的数据。

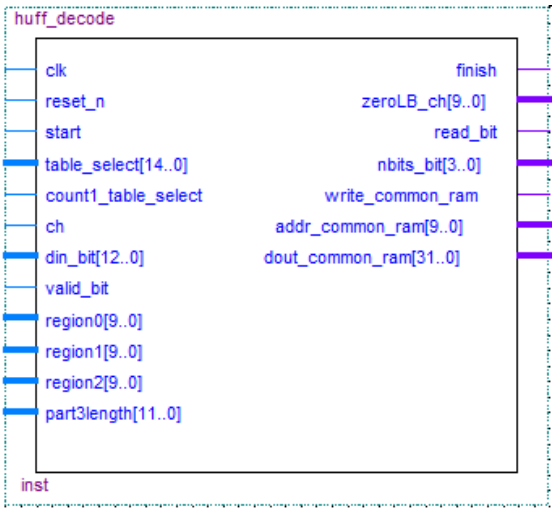

图 5.6 Huffman 解码模块图

Fig.5.6 Chart of Huffman decode module

当 start 为高电平时，Huffman 模块开始工作，valid_bit 有效时，从比特池中读数据，通过 region0，region1，region1 来确定是大值区的值，并由 table_select 选择不同的码表，解完大值区的值后，开始对小值区进行解码，由 count1_table_select 来选择小值区的码表，零值区不用解码。解码完成后将结果写入 RAM 单元中。仿真结果如图 5.7 所示，我们以“ISO/IEC 11172-3”标准中的霍夫曼码表 2 为例进行验证。输出 dout_common_ram 前 4 位表示 x 解码值，后 4 位表示 y 的解码值。实际的解码结果可表示为（x,y）分别是（0,0）（1,0）（0,1）（1,1）（2,0）。

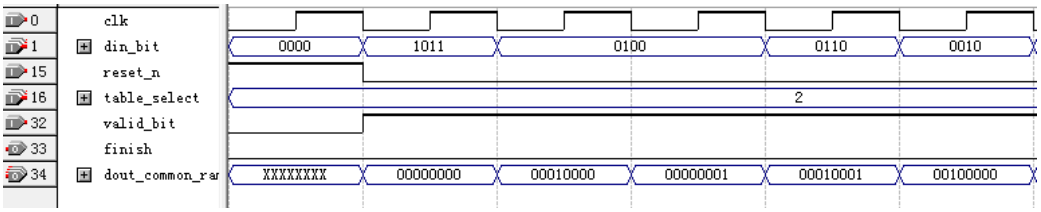

图 5.7 Huffman 解码模块仿真结果

Fig.5.7 Simulation results of Huffman decode module

（3）反量化模块

主要端口说明：

scalefactor：全局比例因子值。

block_type：数据块类型。

men_data：需要输入的数据。

preflag：长块数据，高频预加重。

scalefac_scale：比例因子采用对数量化的形式。

global_gain：全局比例因子的量化步长。

subblock：短块时自带中更细化的量化值。

men_address：输出数据存入地址。

men_writdata：输出要写入内存的整数数据。

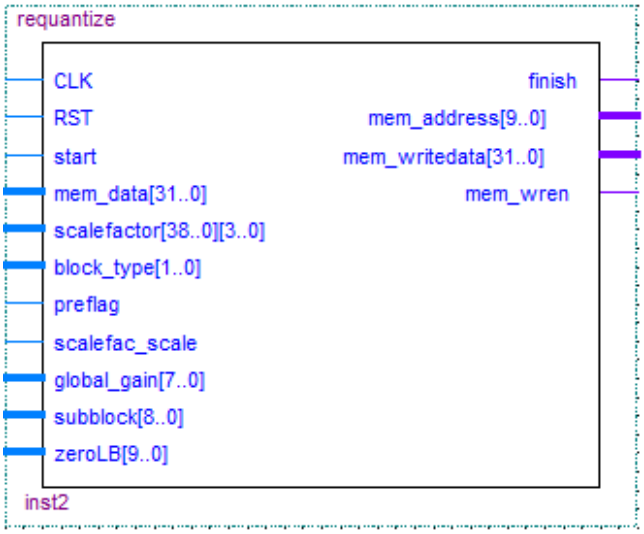

图 5.8 反量化模块图

Fig.5.8 Chart of requantize module

当 start 为高电平时，反量化模块开始工作，通过 men_data 输入数据，并由 block_type 判断数据块类型，由 global_gain 决定比例因子的量化步长与每个采样值相乘，量化步长用二进制表示为 011。以长块类型的数据为例，仿真结果如图 5.9 所示，计算得到的值为浮点数。

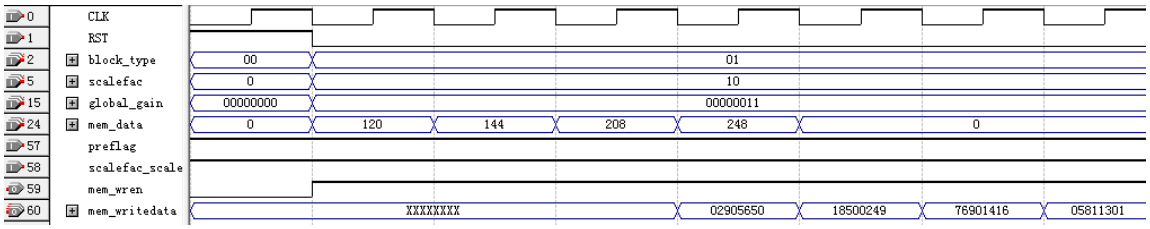

图 5.9 反量化模块仿真结果

Fig.5.9 Simulation results of requantize module

（4）IMDCT 变换模块

主要端口说明：

block_type：确定块类型。

mixed：有混合块类型数据。

cq：输入数据

caddr：输出数据内存地址。

cdata：输出的数据。

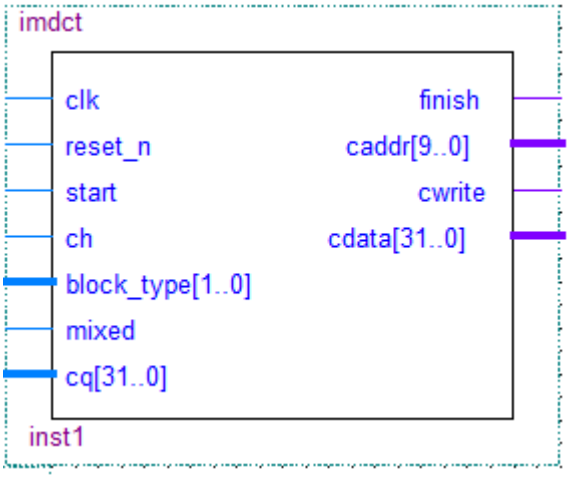

图 5.10 IMDCT 模块图
Fig.5.10 Chart of IMDCT module

当 strat 为高电平时模块开始工作，数据通过 cq 输入，由 block-type 判断块类型，进过一系列计算后，结果通过 cdata 输出，finish 置 1。当数据是短块类型时，仿真结果如图 5.11 所示，输入 6 个值，经过 IMDCT 变换后可得到 12 个结果，经过 3 次加窗结果的叠加，可得到 36 个结果，在前 6 次和后 6 次进行补 0。当数据是长块（起始块）类型时，输入 18 个值，经过 IMDCT 变换后可得到 36 个结果，仿真结果如图 5.12 所示，乘以窗函数后，结果后 6 位为 0。

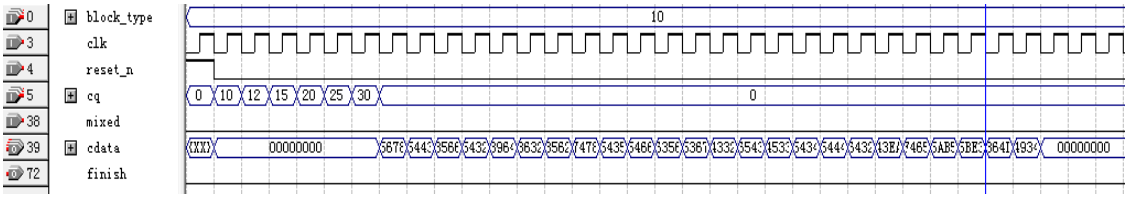

图 5.11 短块块类型加窗后的结果
Fig.5.11 Simulation results of short blocks

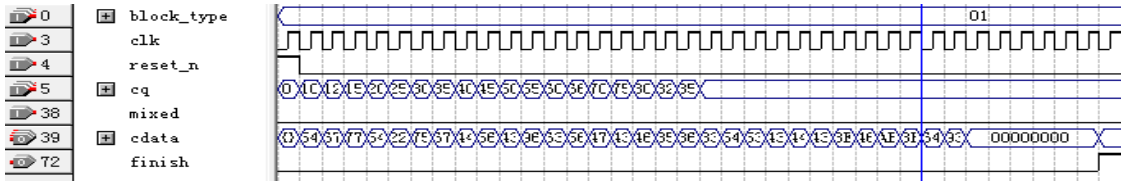

图 5.12 长块类型的加窗后的结果
Fig.5.11 Simulation results of long blocks

§5-2 结果验证及分析

本文在进行 MP3 音频解码器的设计时，使用利用 Quartus II 软件进行简单的功能仿真。但是由于整个设计数据比较繁杂，仅仅依靠仿真图查看设计的正确性是相对比较困难的。因此为了更直观的进行验证，我们利用 Cool Edit 软件（一款功能强大、效果出色的多轨录音和音频处理软件），对解码后的结果和原始文件进行比对。

提取 RAM 单元中的计算结果，与原始音频数据进行比对，便可检验解码功能的正确性。测试的方法是利用 textio 库中的写函数，将 MP3 解码器的子带合成滤波器模块输出的 PCM 数据写入到一个 pcm_samples_from_filterbank.txt 文件中。因为这些数据实际上就是 PCM 文件中的内容，所以将该.txt 文件直接更改后缀名生成文件 pcm_samples_from_filterbank.pcm。再利用 Cool Edit 软件播放该.pcm 文件，查看生成的时域波形图，并与原始波形进行对比，如图 5.13 和图 5.14 所示。

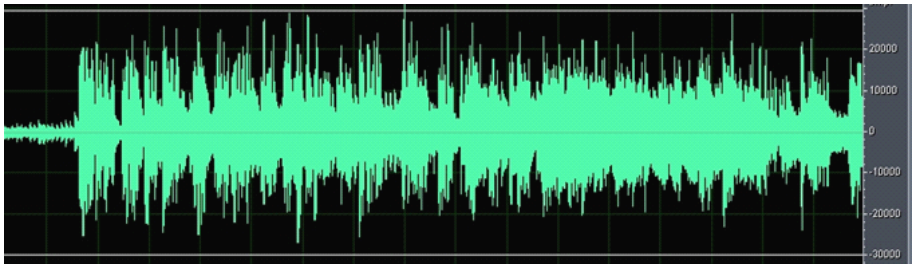

图 5.13 解码后 PCM 数据的时域波形图
Fig.5.13 Time-domain wave form of decoded PCM data

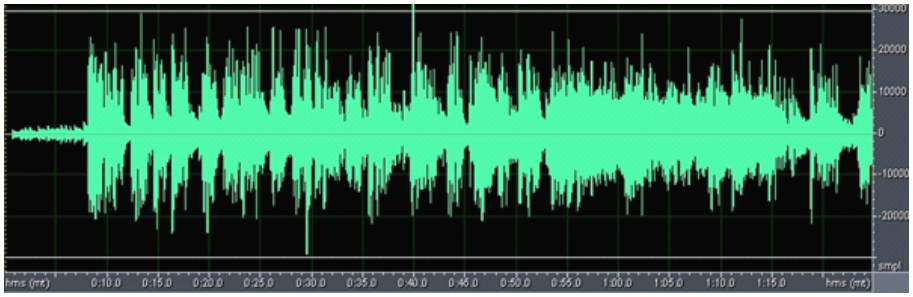

图 5.14 原始 MP3 文件的时域波形图
Fig.5.14 Time-domain wave form of original MP3 file

从两副图中可以看出，二者的波形基本相同，存在的误差较小，这也说明 MP3 解码器的解码功能是能正常工作的，也就是说设计满足要求。

对仿真过程中产生的波形进行时序分析，以长块类型数据为例，得到主要子模块的处理周期数如表 5.1 所示。

表 5.1 子模块解码处理周期及百分比
Table5.1 Submodule decoding processing cycle and percentage

子模块	时钟周期	比例
霍夫曼解码模块	2650	6%
反量化模块	1152	2%
立体声处理模块	1152	2%
混叠重建模块	990	2%
IMDCT 模块	11520	24%
子带综合滤波模块	29952	63%
总计	47416	99%

其中 Huffman 解码模块工作时，因总的码字长度不固定，所以其解码周期数是变化的，在此列举的是一般情况下的处理周期数。IMDCT 模块和子带综合滤波模块几乎占据了 90%的计算量，因此在以后的工作中可以探索更为优化的硬件实现算法，以加快解码速度。

由于本设计对 IMDCT 的算法进行了结构优化，相比较中的其他文献所用的时钟周期，本文的 11520 个周期实现的更快的处理速度。加上参量解析模块的时钟周期，一个颗粒的解码大约为 48576 个时钟周期。对于采样率为 44.1MHz 的 MP3 文件，解码要求一帧需要在 23ms 以内，因此解码电路时钟频率最少是 $48576 \times 2/23\ ms \approx 4.1MHz$,所以当电路频率大于 4.1MHz 时就可以满足实时解码的要求。本次选用的 MP3 格式的文件的采样率是 44.1KHz，实验表明解码功能良好，实现了论文开始时所要求的目标。

第六章 结论

本文的设计对象是 MP3 硬件解码器，设计任务是深入分析 MP3 音频解码算法，以硬件实现该算法为目标，并根据硬件实现的特点，在算法研究的基础上对其进行相关改进和优化，然后讨论各个模块从算法到硬件逻辑结构的映射。通过毕业设计，了解到开发一套功能系统的具体流程，从硬件平台的构建，到软件驱动的编写，再到最后系统的整合和调试，这些都是开劣系统的必要流程。在专业知识方面，对 FPGA 设计和 MP3 解码算法以都有了深入的研究。

在方案论证阶段，查阅了大量与 MP3 播放器和解码器相关的期刊文献，对各种方案进行论证，分析出各自的特点，最后确定基于 FPGA 实现 MP3 播放器的方案，这样可以提高解码速度，并降低系统消耗。

在方案设计时，先从总体上进行功能分析，然后进行各项功能的需求分析，再在此需求分析的基础上进行系统的总体设计。并且将系统模块化，尽量做到细分，在具体实现的时候就相对容易些。分别对系统的各个模块进行详细设计与实现。先实现硬件构件，再编写驱动程序对其进行控制，这样对每个模块进行验证。

本文分别对各个模块进行了基本的功能仿真，结果是令人满意的，该设计完成了 MP3 播放器的基本功能，在提高解码速度和降低系统消耗方面都具有自己的优势。

不足之处在于，只是用于 44.1KHz 的单一频率，而且由于本文用的是简单的双声道，因此对立体声模块没有进行数据处理。在以后的工作中可以对这方面加以改进。

参考文献

[1]张林林.基于压缩域特征的视频检索技术研究[D].北京:北京交通大学,2010

[2]张兵.视频会议中的视频数据 MPEG 处理[J].信息技术,2002, 21(12):48-50

[3]侯东京.MPEG-1音频第Ⅲ层编码器的研究与设计[D].南京:南京理工大学,2004

[4]杜福慧.MP3音频解码器的 FPGA 原型芯片设计与实现[D].合肥:合肥工业大学,2009

[5]宋奇刚,魏小义.霍夫曼解码器的设计及在 MP3 解码中的应用[J].今日电子,2005,8 (03):33-34

[6]吕琛,王小雪,杨会成.MP3 解码的优化及实现[J].合肥工业大学学报,2011,5(08):18-21

[7]陈建寿,陈咏恩.MP3 解码的 IMDCT 优化算法[J].信息安全与通信保密,2008,(10):43-45

[8] Britanak V. An Efficient Implementation of the Forward and Inverse MDCT in EG Audio Coding.IEEE Signal Processing Letters,2001,Vol.8 (No.2):48-51

[9]吕琛.基于 FPGA 的数字音频编解码系统的研究[D].芜湖:安徽工程大学,2010

[10]刘华,刘卫东,邢文峰.基于软硬件协同设计的 Huffman 解码模块[J]电子设计工程,2011(06):25-26

[11] Takala J,Rostrom J,Vaaraniemi T,Herranen H. A Low-Power MPEG Audio Layer III Decoder IC with an Integrated Digital-to-Analog Converter.IEEE Transactions on Consumer Electronics, 2000,46(3):896-902

[12]陈伦艳.基于 ARM9 的网络 MP3 播放器的研究与实现[D].大连:大连理工大学,2008

[13]李菁菁.MP3 软件解码器的研究与实现[D].大连:大连海事大学,2006

[14]丰帆.MP3 数字音频编解码算法的研究及实现[D].成都:西安电子科技大学,2008

[15]陈艳,赵歆,李平.基于硬件描述语言的 MP3 解码器仿真平台的搭建以及 IP Core 的重用[J].现代电子技术,2004,10(15):48-51

[16] Bhasker .Verilog HDL Synthesis A Practical Primer[M] .USA: Star Galaxy Pub, 1998 Vol.10(No.5):136-143.

[17]尹丹.基于 IP 网络的嵌入式音频系统设计与实现[D].长沙:中南大学,2011

[18]伍汉华. MP3 数字音频编解码的研究及 IMDCT 模块的硬件实现[D].成都:电子科技大学,2005

[19]郑鹏.基于 FPGA 的时差法超声波流量计系统的设计与实现[D].杭州:浙江大学,2006

[20]王幸.音频编解码算法的研究与实现[D].成都:成都理工大学,2009

[21]陈立军.基于 OMAP1510 处理器 MP3 解码的研究与实现[D].长春:长春理工大学,2005

[22]袁伟.基于 ARM9 的 MP4 设计[D].武汉:武汉科技大学,2009

[23]王宏武.基于 ARM7 嵌入式系统的 MP3 设计[D].天津:天津大学,2009

[24] Shamshiri S.,Fakhraie S. Parallel Alias Reduction for MP3 Decoding[C].IEEE International Conference on Microelectronic,2004,Vol.10(No.5):438-441

[25]陆志洋.MPEG-1Audio Lay3 解码器实现及应用[D].大连:大连海事大学,2006

[26]张曦.嵌入式 MP3 播放器的研究与实现[D].上海:复旦大学,2004

[27]张磊.基于 MIPS 平台的 MPEG-4 AACplus v2 解码器的实现[J].电声技术,2007(12):18-21

[28]王承峰.基于 ARM720T 微处理器的 MP3 解码算法实现[D].南京:南京航空航天大学,2001

[29]龚红波.数字电视机卡分离技术的研究与应用[D].武汉:华中科技大学,2007

[30]彭新生.MP3 音频解码算法研究及单片 VLSI 硬件解码实现设计[D].武汉:华中师范大学,2000

[31]张树刚,张遂南.CRC 校验码并行计算的 FPGA 实现[J].计算机技术与发展,2007,17(2):56-58

[32]苏祖辉.IMDCT 在 MPEGAudio-1 LayerⅢ中的递归实现[D].合肥:合肥工业大学,2005

[33] Anguita,M.,Martinez-Lechado J.M. MP3 optimization exploiting processor architecture and using better algorithms.IEEE Micro[J],2005,25 (3):81-92

[34]杨小牛,楼才义,徐建良.软件无线电原理与应用[M].北京:电子工业出版社,2006

[35]潘松.SOPC 技术实用教程[M].北京:清华大学出版社,2005

[36]潘松,黄继业.EDA 技术与 VHDL(第 2 版)[M].北京:清华大学出版社,2008:107-112

[37]董代洁,郭怀理.基于 FPGA 的可编程 SOC 设计[M].北京:北京航空航天大学出版社,2006:220-228

[38]孙红英.语谱分析的 FPGA 实现[J].电子与信息学报,2011,33(5):89-93

[39] Byeong Lee.A new algorithm to compute the discrete cosine Transform.IEEE Transactions on Signal Processing,1984,32(6):1243-1245

[40]刘明业,蒋敬旗,刁岚松.硬件描述语言 Verilog[M].北京.清华大学出版社,2001:156-189

[41]周锦锋.低功耗 MP3 解码器设计及其可测性分析[D].中国科学院计算技术研究所,2003,(07)

[42]于红旗,仲涛.TEXTIO 及其在 VHDL 仿真中的应用[J].单片机与嵌入式系统应用,2004,3(2):81-83

[43]赵明富,李太福.Linux嵌入式系统实时性分析与实时化改进[J].计算机应用研究,2004,12(5):43-45

[44]窦维蓓,王建昕,董在望.基于DSP的IMDCT快速算法[J].清华大学学报,2000,7(2):34-35

[45]褚振勇,翁木云.FPGA设计及应用[M].西安电子科技大学出版社,2002:229-238

[46] Wang Z,Jullien G,Miller W.Recursive Algorithms for the Forward and Inverse Discrete Cosine Transform with Arbitrary Length. IEEE Signal Processing Letters, 1994,7(1):101-102

[47]贺敬凯.Verilog HDL数字设计教程[M].西安电子科技大学出版社,2010:129-144

[48]王晓明. Verilog HDL程序设计与实践[M].人民邮电出版社,2009:57-66

[49]刘艳萍,高振斌,李志军.EDA实用技术及应用[M].国防工业出版社，2008:175-186

[50]孙航.Xilinx可编程逻辑器件的高级应用与设计技巧[M].北京:电子工业出版社,2004:156-172

[51]徐欣,于红旗等.基于FPGA的嵌入式系统设计[M].北京:机械工业出版社,2005:243-258

[52] Kohn Leslie,Rice Daniel.MPEG video decoding with the ultra SPARC visualin struction set[J].IEEE Computer Society Int-ernational conference,1995:470-475

[53]李成奇.基于FPGA技术的视频采集系统设计及实现[D].哈尔滨:哈尔滨理工大学,2008

[54]王道宪.CPLD/FPGA的可编程逻辑器件应用与开发[M].北国防工业出版,2004:75-78

[55]俞莉琼.MPEG LayerⅢ音频解码器中混合滤波模块的口设计[D].上海:上海交通大学,2005

[56]陈伟宁,秦士.多相滤波器的原理及实现[J].清华大学学报(自然科学版),2001,4(1): 9-11

附录

主要模块的部分代码如下：

1. 取主数据模块

```
always @(posedge clk, negedge reset) begin
    if ( !reset)begin
        state <= IDLE;
        count <= 12'b0;
        fifo_clear = 1; //fifo empty at reset
    end
    else begin
        write_data = 1'b0;
        rdreq = 1'b0;
        finish = 1'b0;
        fifo_clear = 0;
        case(state)
            IDLE:    if(start)begin
                        state <= SHIFTOUT;
                        count <= 0;
                    end

            SHIFTOUT:begin
                        if(main_begin_data==0) begin
                            fifo_clear = 1;              //flush out all data in fifo
                            state <= FIRST_GETDATA;
                        end
                        else if( usedw > main_begin_data )
                            state <= FIFO_OUT;
                        else if( usedw + 3 < main_begin_data )
                            state <= FILL_ZERO;
                        else
                            state <= FIRST_GETDATA;
                     end

            FIFO_OUT:begin
                        rdreq <= 1'b1;
                        if( usedw == main_begin_data + 1)
                          begin
                          rdreq <= 1'b0;
                            state <= FIRST_GETDATA;
                        end
                    end
```

```
                FILL_ZERO:begin // not actually zero
                            write_data<=1'b1;
                            if (usedw == main_begin_data) begin
                                write_data<=1'b0;
                                state <= FIRST_GETDATA;
                            end
                        end

                FIRST_GETDATA :
                        begin
                            count <= count + 1;
                            state <= GETDATA;
                        end
                GETDATA:begin
                            write_data<=1'b1;
                            if(count < lengthframe)
                                count <= count + 1;
                            else
                                state <= DONE;
                        end

                DONE:   begin
                            finish <= 1'b1;
                            state <= IDLE;
                        end
                default  : ;
            endcase
        end
end
endmodule
```

2.Huffman解码模块

```
begin
   case({r2,r1})
      2'b11: select     <= table_select[4:0];          // in region 2
      2'b01: select     <= table_select[9:5];          // in region 1
      2'b00: select     <= table_select[14:10];        // in region 0
      default: select <= table_select[4:0];
   endcase
end
begin
   case(select)
      0 : base_addr = Table_Base0; //0
      1 : base_addr = Table_Base1; //1
      2 : base_addr = Table_Base2; //2
```

```
        3 : base_addr = Table_Base3; //3
        4 : base_addr = Table_Base4; //4      not used
        5 : base_addr = Table_Base5; //5
        6 : base_addr = Table_Base6; //6
        7 : base_addr = Table_Base7; //7
        8 : base_addr = Table_Base8; //8
        9 : base_addr = Table_Base9; //9
        10: base_addr = Table_Base10; //10
        11: base_addr = Table_Base11; //11
        12: base_addr = Table_Base12; //12
        13: base_addr = Table_Base13; //13
        14: base_addr = Table_Base14; //14    not used
        15: base_addr = Table_Base15; //15
        16, 17, 18, 19, 20, 21, 22, 23: //16,17,18,19,20,21,22,23 = {1,2,3,4,6,8,10,13}
           base_addr = Table_Base16;
        24, 25, 26, 27, 28, 29, 30, 31: //24,25,26,27,28,29,30,31 = {4,5,6,7,8,9,11,13}
           base_addr = Table_Base24;
    endcase
end
c_start: begin
            if(ctable == quadTableA)              /*Count1 table is quadTable A,decode as above*/
                hstate       <= c_readHuffCode;
            else if (remainBits < 4)
                hstate       <= z_start;
            else if (bvalid) begin                /*Count1 table is quadTableB, much more simple*/
               hstate       <= hwait;
               hnext         <= c_simple;
               wcount        <= 0;

               bnbits       <= 4;
               bread         <= 1'b1;
            end
            first       <= 1'b1;
            countSignBits <= 2'b00;
         end

         c_read1Bit:
         if (OutofBits)
            hstate       <= z_start;
         else if(bvalid) begin
            hstate      <= hwait;
            hnext        <= c_readHuffCode;
            wcount      <= 0;//wait 1 clock cycles

            bread       <= 1'b1;
```

```
    bnbits      <= 1;
end

c_readHuffCode: begin
    hstate     <= hwait;
    hnext      <= c_chkLeaf;
    wcount     <= 2;
    haddr      <= first?{9{1'b0}} : {bdin[0] ? hdin[ 8: 0] : hdin[17: 9]};
    first      <= 1'b0;
end

c_chkLeaf: begin /*check leaf for V, W, X and Y*/
    if(hdin[17: 9] == {9{1'b0}}) begin
        hstate        <= c_signBits;
        dataVWXY      <= hdin[3:0];
    end
    else begin
        hstate        <= c_read1Bit;
    end
end

c_simple: begin
    hstate          <= c_signBits;
    dataVWXY        <= {~bdin[3],~bdin[2],~bdin[1],~bdin[0]};
end

c_signBits:
if (OutofBits) begin
    hstate      <= z_start;
    if(countSignBits[0] | countSignBits[1])
          caddr <= caddr + 1'b1;
end
else if(bvalid) begin
    unique case(countSignBits)
        2'b00: bread <= dataVWXY[3];
        2'b01: bread <= dataVWXY[2];
        2'b10: bread <= dataVWXY[1];
        2'b11: bread <= dataVWXY[0];
    endcase
    bnbits             <= 1;
    hstate             <= c_delay1;
    if(countSignBits[0] | countSignBits[1])
        caddr <= caddr + 1'b1;
end
```

```
            c_delay1: begin
               hstate              <= c_storeVWXY;
            end
```

3.反量化模块

```
    always_ff @(posedge CLK, negedge RST ) begin
        if ( !RST )
            state <= IDLE;
        else
            state <= next_state;
    end
        always_comb begin
        next_state = state;
        mem_wren = 1'b0;
        finish = 1'b0;
        unique case ( state )
            IDLE            :   if ( start ) next_state = READ;
            READ            :   next_state = READDELAY1;
            READDELAY1      :   next_state = READDELAY2;
            READDELAY2      :   next_state = CALCULATE;

            CALCULATE       :   next_state = CALCULATE1;
            CALCULATE1      :   next_state = WRITE;
            WRITE           :   begin next_state = LOOP; end

            LOOP            :   begin
                                    //if ( mem_address < 576 )
                                    if ( mem_address < zeroLB ) begin
                                        mem_wren = 1'b1;
                                        next_state = READ;
                                    end
                                    else
                                        next_state = DONE;
                                end
            DONE            :   begin next_state = IDLE; finish = 1'b1; end
            default         :   ;
        endcase
    end

    logic [31:0] is;
    always_ff @(posedge CLK, negedge RST) begin
        if ( !RST ) begin
            is <= 0;
        end
        else begin
            unique case ( state )
                IDLE            :   begin mem_address <= 0;     mem_writedata <= 0; end
```

```
                READDELAY2    :    is <= mem_data;
                WRITE         :    mem_writedata <= xr;
                LOOP          :    mem_address <= mem_address + 1;
                DONE          :    ;
                default       :    ;
            endcase
        end
    end
```

4.IMDCT 变换模块

```
i_accum: begin
        multA      <= Xk;
        multB      <= cosq;
        k          <= k + 1'b1;
        addA       <= multP;
        addB       <= addS;

        if(k == (acc_num + 2'd3)) begin
          i_state<= i_window;
          i        <= i + 1'b1;
          k        <= 0;
        end
      end
       i_window: begin
        i_state  <= i_store;
        multA      <= addS;
        multB      <= wq;
        m          <= m + 1;
      end
      i_store: begin
        i_state          <= i_accum_delay1;
        im[index]<= multP;
        addA             <= 0;
        addB             <= 0;
        multA            <= 0;
        multB            <= 0;
        if(isShort) begin
          if(i == 12 || i == 24) begin
            caddr   <= cnext + 1'b1;
            cnext   <= cnext + 1'b1;
            i_state   <= i_readInfo_delay1;
            m        <= 0;
          end
        end

        if(i==36) begin
          if(isShort)begin
```

```
                caddr       <= cnext - 3;          //reset the addr for writing back
                cnext       <= cnext + 16;
                i_state    <= i_overlap_short1;
                k              <= 12;
             end
             else begin
                caddr       <= cnext-1;           //reset the addr for writing back
                cnext       <= cnext + 5'd18;
                i_state    <= i_overlap_delay1;
             end
          end
       end
```

5.子带综合滤波模块

```
s_dct: begin//i can use a mult_acc, but i just use a mult and an add seperately
          Si[k] <= cq;
          caddr <= caddr + 18;
          k <= k + 1;
          if ( k == 31 ) begin
                state <= s_dct_acc;
                k <= 0;
          end
       end

       s_dct_acc: begin
          k         <= k + 1'b1;
          multA <= Ni;
          multB <= Si[k];

          addA <= multP;
          addB <= addS;
          if(k == 31) begin
                state        <= s_write1024;          //write to buffer 1024
                caddr[9:0]       <= s[4:0] - 18 ;       // add = s*32 for writing PCM
                k <= 0;
          end
       end
       s_write1024: begin                    //write 64 new values to buffer 1024
          //we need a [5:0] c (counter) to count the number of values written
          c       <= c + 1'b1;
          //reset the accumulator
          addA    <= 32'd0;
          addB    <= 32'd0;
          multA <= 32'd0;
          multB <= 32'd0;

          b1024_write         <= 1'b1;
```

```
    b1024_addr          <= {base1024[ch][3:0], c[5:0]};
    b1024_data          <= addS;//data must be ready

    if(c == 6'd63)
        state     <= s_PCM_delay1;
    else
        state <= s_dct_acc;
end
```

致谢

这次能顺利地完成我的论文，要感谢研究生期间给过我帮助的所有老师、同学和朋友，还有一直在我背后默默支持我，关心我的最亲爱的家人，他们始终是我最坚强的后盾。

第一个要感谢的是我的指导老师——田学民老师，两年前我幸运地称为他的第一个弟子。田老师广阔的眼界，渊博的学识，宽以待人的处事作风时刻激励着我们奋发前进。田老师在繁忙的事物之中总能抽时间指导我的工作学习，认真批阅我的论文，在此，向田老师表示深深的感谢。

此外，最让我感念的是在我做课题期间，田老师帮我解决了很多问题，他认真的批阅我的论文，可以说我的每一个进步都离不开老师的心血。在老师的鼓励下，利用假期在公司实习，这次实习的经历让我对马上就要进入的社会有了更深刻的体会。我不会辜负老师的期望，会坚持自己的目标，努力实现自己的梦想。

其次要感谢的是我的师弟王志欣和师妹尚君萤，还有我始终如一的朋友们。感谢师弟对我无私的帮助，在实习期间由于不能及时回到学校，师弟给了我太多的帮助，是他让我体会到了同门师姐弟的温暖，对此我将铭记在心。还有在我最痛苦和无助的时候，给我最多支持和关怀的朋友们，六年的时间换了我们的友谊会地久天长。

最后要感谢我亲爱的家人，我的父母还有我可爱的妹妹们，正是你们的关怀，才让我勇敢地走到今天，你们永远是我前进的动力。

攻读学位期间所取得的相关科研成果

[1] 田学民,张晓境. MP3 音频解码优化算法[J].电子设计工程（已录用待发表）。