基于 FPGA 的 LZ4 无损压缩算法优化设计

摘 要

随着云计算和大数据产业的兴起，计算机系统对数据存储和传输速率要求越来越高，现有的无损压缩软件逐渐变得难以满足实时数据访问的速率需求。新一代 LZ4 算法得益于百兆字节每秒（Million Bits per second，MBps）级别的压缩速率，在高速压缩场合得到广泛的应用。专用硬件压缩电路能够充分利用硬件的并发性和实时性，提供较高的压缩率和压缩速率性能，并且，压缩过程不需要占用中央处理器（Central Processing Unit，CPU）的计算资源，对压缩率和压缩速率性能之间的折中只取决于电路结构。

本文提出了针对 LZ4 压缩率缺陷的优化方案。使用现场可编程门阵列（Field Programmable Gate Array，FPGA）设计并实现了字典缓冲器、并行匹配电路、字符串分割电路、并行编码器、校验电路和流水线控制器，共同组成 LZ4 压缩电路。为了进一步优化压缩率，提出了以半静态哈夫曼（Huffman）编码为基础的二级压缩方法，并使用 FPGA 设计了统计、排序、建树、码长优化、码表生成以及编码电路。此外，将 LZ4 电路和半静态 Huffman 编码电路进行级联，解决了 LZ4 压缩电路的压缩率与兼容性之间的矛盾。

本文所述的压缩电路在 Xilinx KC705 开发平台上进行测试。设定电路工作频率 125MHz，使用卡尔加里语料库（Calgary Corpus）和坎特伯雷语料库（Canterbury Corpus）进行性能测试。结果表明，在兼容模式下，平均压缩率（52.76%和 49.95%）和压缩速率（213.09MBps 和 217.93MBps）基本达到 LZ4 压缩软件的水平；在优化模式下，设置 16 千字节（Kilo Bytes，KB）的统计长度能够兼顾压缩率和压缩速率性能，平均压缩速率分别为 185.72MBps 和 172.66MBps，平均压缩率（44.89%和 42.60%）性能相对 LZ4 软件提升 14%以上。在压缩速率损失较少的前提下，有效的提升了压缩率性能，体现 LZ4 电路可以灵活的在压缩率与压缩速率之间进行折中的优势。

关键词：无损数据压缩，LZ4 算法，Huffman 编码，压缩率优化，FPGA 硬件实现

目录

第一章 绪论

1.1 课题背景与意义

自 20 世纪 70 年代以来，计算机和互联网的普及为人类的生产生活带来了极大的便利。与此同时，存储数据量的快速增长和传输速率的不断提高，使数据压缩技术开始得到广泛的关注和应用。根据压缩过程是否丢失信息，数据压缩被分为有损压缩和无损压缩两大类，每大类又包含多种不同的压缩算法。有损压缩一般针对特定的形式的信息，例如图像、视频、音频等。而无损压缩可以用于任意信息的压缩，包括文本、网页[错误!未找到引用源。]、图像[错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。]、专用数据流等[错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。]，是相对有损压缩更为通用的数据压缩方法。

无损压缩的关键是利用等信息量短数据替换长数据的方法消除重复的信息，降低待压缩数据中的冗余，并保证替换后没有信息被丢失。于是，依据不同的替换原理，将无损压缩分为字典类、统计类和预测类三种算法。其中最常用的是字典类压缩算法，起源于 1977 年两位以色列科学家 Jacob Ziv 和 Abraham Lempel 提出的 LZ77 算法[错误!未找到引用源。]。图 1-1 列举了每类对应的压缩算法示例。LZ4 属于字典类无损压缩算法，即用字典中匹配数据的偏移量、长度、剩余字符等短数据来替换原始的长数据，达到压缩数据长度的目的。霍夫曼（Huffman）编码属于统计类，依据信源概率对每个字符生成对应的变长编码（Variable Length Code，VLC），再依次替换数据中的所有字符，实现用短编码替换出现概率高的字符，用长编码替换出现概率低的字符，从而降低整体的数据长度。

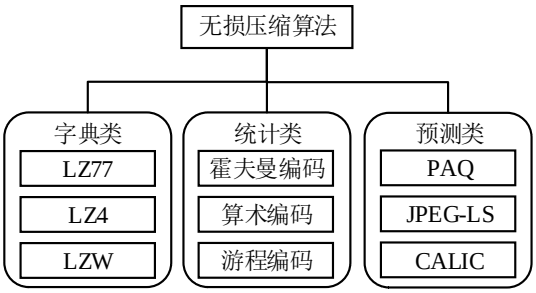


图 1-1 无损压缩算法分类和示例

随着云计算和大数据产业的兴起，计算机系统中对数据存储、传输速率和实时性要求越来越高，现有的无损压缩软件逐渐变得难以满足实时数据访问的速率需求[错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。]。于是，注重压缩速率的 LZ4、LZO、QuickLZ 以及 Snappy 等算法开始受到人们的重视。其中，LZ4 压缩算法已被应用于海量数据存储系统[错误!未找到引用源。]和高清图像[错误!未找到引用源。]的实时数据压缩。其特点是压缩和解压缩的速率远高于现有的压缩算法[错误!未找到引用源。]，对一般数据源的压缩速率可达到数百兆字节每秒（Million Bits per second，MBps），非常适用于对压缩和解压缩速率很敏感的实时无损压缩应用场合。然而，这些算法的高压缩速率是以牺牲压缩率为代价的，所以一般情况下它们的压缩率并不理想。

以 Huffman 编码为首的统计类编码具有输出平均码元长度无限逼近信源熵的特点[错误!未找到引用源。]，已经在各类压缩软件中得到了广泛的应用。为了获得较好的压缩率，高性能无损压缩算法通常由两级压缩组合而成，第一级为字典类或预测类压缩，第二级为统计类压缩。例如，针对图片的 PNG [错误!未找到引用源。]和针对文本的 Gzip [错误!未找到引用源。错误!未找到引用源。]都是由 LZ77 算法和 Huffman 编码构成的；针对文件系统的 7-ZIP 软件在默认情况下由 LZMA [错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。]算法和算术编码[错误!未找到引用源。]构成。

目前，无损压缩算法主要使用计算机软件实现。受制于软件串行执行、过度依赖计算机中央处理器（Central Processing Unit，CPU）计算资源的局限性，压缩软件压缩速率较低、占用 CPU 资源多、压缩实时性差，逐渐成为系统的瓶颈。而专用无损压缩电路可以利用并发结构来提升压缩速率，并减少对其他计算资源的依赖，通过设计不同的硬件结构，使电路性能在压缩率和压缩速率之间进行灵活的折中。

针对无损压缩算法的研究已经持续了近 40 年，各类无损压缩软件被广泛的应用于存储和传输领域。然而，针对无损压缩电路的研究却并不多。近十年来，专用集成电路（Application Specific Integrated Circuit，ASIC）和现场可编程门阵列（Field Programmable Gate Array，FPGA）的性能逐步提升，使得人们开始重视无损压缩算法的硬件实现方面的研究。同时，各种处理速率或效率更高的无损压缩算法及其对应的软件[错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。]、硬件架构[错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。]或处理单元[错误!未找到引用源。错误!未找到引用源。]相继提出，逐渐被广泛使用。

1.2 国内外研究现状

随着计算机网络传输速度的提升，国内外研究机构和高校对 LZ4 无损压缩算法和硬件压缩电路的研究正逐渐增加。而 LZ4 压缩软件过度依赖 CPU 计算资源，从而使应用环境受到局限，经济性也不好。

由于 LZ4 是一种运行在通用计算机环境下的开源软件压缩算法，所以关于 LZ4 算法的研究首先在压缩软件方面得到应用。例如，同济大学超大规模集成电路研究所提出了一种基于 LZ4HC 算法的高清图像压缩软件[错误!未找到引用源。]，软件运行需要高性能 CPU 支持；巴西的研究机构针对原始格式的超高清（Ultra High Definition，UHD）视频提出了基于 LZ4 的专用软件无损压缩方案[错误!未找到引用源。]，为了满足视频流的数据传输速率要求，该方案使用了高性能服务器中的 16 个 CPU 同时进行压缩，消耗了大量的计算资源。

为了减少压缩过程对通用计算资源的消耗，捷克的一所高校研究了一种 LZ4 压缩电路在超高清视频系统上的实现和应用[错误!未找到引用源。]，但该电路的数据处理宽度仅 8 位（bit），即每次处理 1 个字节的数据，导致压缩速率的瓶颈比较突出，并且，电路实现对 LZ4 算法本身也做了简化，使压缩率变差。

韩国首尔大学研究了一种针对外汇交易数据的硬件压缩加速平台[错误!未找到引用源。]，利用两个 LZ4 压缩核实现了压缩电路，单核压缩速率仅 75MBps。每个压缩核内部有 20 级流水线，其中匹配模块独占 17 级，处理效率较低，导致其压缩速率难以提高，没有充分发挥硬件电路的并行处理优势。

由于 LZ4 属于字典类无损压缩算法，其去冗余的方法与其他字典类算法类似，而针对现有字典类无损压缩算法的硬件电路的研究相对较多，例如东南大学和滑铁卢大学分别使用 Xilinx ML605 和 Altera DE2 FPGA 开发板实现了 Gzip 压缩电路[错误!未找到引用源。错误!未找到引用源。]，东南大学和印第安纳大学分别使用 Xilinx ML605 和 Xilinx Spartan 3E 开发板实现了 LZMA 压缩电路[错误!未找到引用源。错误!未找到引用源。]，台湾大学设计了基于 0.35um 工艺的 LZW 压缩电路[错误!未找到引用源。]。上述无损压缩电路都存在压缩速率偏低，硬件实现时压缩算法被简化，牺牲了压缩率性能等缺陷。

在 Huffman 编码电路方面，印度的一所工程学院提出了静态 Huffman 编码的 FPGA 电路实现[错误!未找到引用源。]，使用固定的变长编码表，将待压缩字符直接替换，达到压缩的目的。优点是查表替换进行处理的速度很快，缺点是由于该 Huffman 编码电路针对 LZ77 压缩输出的三元组格式进行压缩，无法压缩其他格式的数据。

滑铁卢大学 Rigler 设计的 Gzip 压缩电路[错误!未找到引用源。]中，采用了动态 Huffman 编码，获得了与 Gzip 软件一致的压缩率，但在 FPGA 上验证的压缩速率比 Gzip 软件慢 7 倍以上。并且，该动态 Huffman 编码电路针对 LZ77 的三元组输出格式，无法用于其他格式数据的编码，通用性较差。

综上所述，当前国内外对 LZ4 无损压缩的研究还处于初级阶段，解决方案还不成熟，压缩软件缺乏经济性，压缩电路的压缩性能方面有较大的提升空间。并且，Huffman 编码电路通常针对专门格式的数据，且处理速率性能较差，有待改进。

1.3 研究内容与设计指标

1.3.1 研究内容

本文旨在利用硬件电路的并行特点，研究一种优化 LZ4 无损压缩的方法，使其在具备较高压缩速率的前提下，提升压缩率性能，改善 LZ4 压缩率差的问题。并且，在(FPGA 平台上)设计相应的硬件电路结构，使电路既可以兼容 LZ4 软件的编码格式，又可以提供压缩率性能更好的工作模式。

主要研究内容包括：

(1) 研究 LZ4 无损压缩算法的关键步骤、输出格式和校验算法，分析各步骤中对压缩率和压缩速率性能有影响的部分，并提出针对性的优化方法，对改进后的 LZ4 无损压缩算法进行硬件电路实现；

(2) 研究 Huffman 编码，并对动态 Huffman 编码流程进行深入分析，针对编码过程中的缺陷提出相应的优化方法，设计经过优化的 Huffman 编码电路，并提供可以级联的接口；

(3) 使用硬件描述语言（Verilog Hardware Description Language，Verilog HDL）完成上述两级压缩电路的寄存器传输级（Register Transfer Level，RTL）设计，并进行逻辑仿真；

(4) 搭建板级实验平台，使用快速外设部件互联总线（Peripheral Component Interconnect Express，PCIe）

接口通过直接存储器访问（Direct Memory Access，DMA）的方式实现 FPGA 内的压缩电路与上位机内存之间的数据传输；

(5) 使用卡尔加里语料库（Calgary Corpus）和坎特伯雷语料库（Canterbury Corpus）两个测试集对设计的无损压缩电路进行测试和验证。

1.3.2 设计指标

本文的主要目标是针对压缩率优化 LZ4 无损压缩算法，并设计相应的无损压缩电路。同时保证经过优化的 LZ4 算法及其压缩电路的输出数据格式能够与原始的 LZ4 压缩软件相兼容。

主要的功能和性能指标包括：

(1) 设计一种基于 LZ4 算法且具备对任意形式数据进行无损压缩功能的压缩电路；

(2) 设计具备兼容模式和压缩率优化模式的压缩电路；

(3) 压缩电路在优化模式下的压缩率性能相比 LZ4 压缩软件性能（对 Calgary 语料库的压缩率约 54%）提升 10%以上；

(4) 在任意模式下，LZ4 无损压缩电路的处理效率不低于 0.2Byte/Cycle；

(5) 在 Xilinx Kintex-7 KC705 FPGA 开发平台上实现上述压缩电路，并且工作频率不低于 100MHz。

1.4 本文组织结构

本文共包含 6 个章节：

第一章为绪论，分析了现有的压缩，尤其是无损压缩技术的背景与意义，并且着重介绍了 LZ4 无损压缩的国内外研究现状，以及本文将要完成的工作；

第二章研究了原始的 LZ4 压缩算法和 Huffman 编码，着重讨论了 LZ4 算法中可能导致性能降低的压缩步骤，以及动态 Huffman 编码的缺陷和优化方法；

第三章阐述了 LZ4 无损压缩电路的总体设计和各模块子电路的实现方法；

第四章阐述了 Huffman 编码电路的架构设计，并着重描述了各模块子电路的实现方法和工作流程；

第五章搭建硬件测试平台，依据标准压缩测试源对整体压缩电路进行实现和测试，并分析测试结果；

第六章为总结和展望，对全文进行系统性总结，并展望本文所述的无损压缩电路在未来的应用场合和改进方向。

第二章 LZ4 压缩和 Huffman 编码算法分析

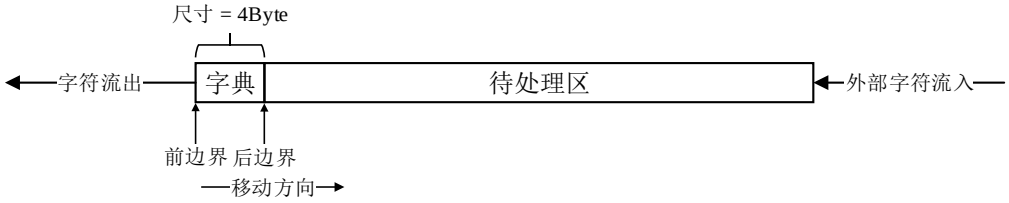
2.1 LZ4 压缩算法分析

LZ4 是由 Y. Collet 编写的一种基于字典匹配思想的开源压缩算法，其特点是压缩速率远高于现有的同类压缩方法[错误!未找到引用源.]。由于 LZ4 是 LZ77 压缩算法的变体，其核心的去冗余方法均可以简化为一系列的查字典、匹配、替换操作。然而，LZ4 压缩算法为了获得很高的压缩速率，在实现上述各操作的过程中采用了不同于传统 LZ77 压缩算法的方式。需要注意的是，LZ4 压缩算法处理的最小单位是 1 字节（Byte）或 8 位（bit）字符，本文中提及的所有字符均代表 1Byte 或 8bit 数据。

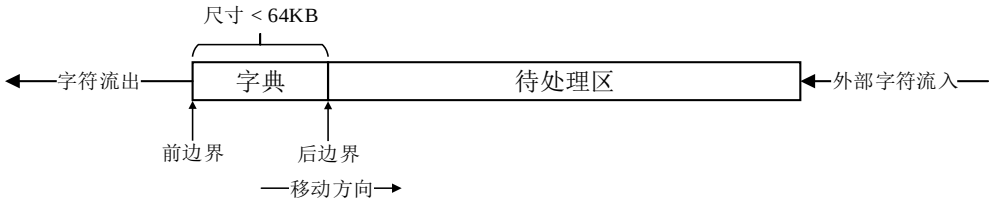
2.1.1 LZ4 压缩的关键步骤

查字典是基于字典匹配压缩算法实现的第一步。假设 LZ4 压缩处理的对象是一个数据宽度为 1Byte 的外部字符流，直接进入缓冲区，在压缩处理过程中，缓冲区可以分为三个部分，分别为已处理区、字典和待处理区。如图 2-1（a）所示，在初始状态下，首先将字典和已处理区尺寸设置为 0，字典的前后边界均处于缓冲区开始处，由于数据流入的最初 4 个字符不存在匹配，将它们直接放入字典中，此时字典尺寸为 4Byte，如图 2-1（b）。然后从紧邻字典后边界的待处理区中获取 1 个字符，与字典中的后 3 个字符拼接成一个新的四字符串，并尝试在字典中搜索与之完全一致的字符串。如果找到了一致的四字符串，则进入匹配操作，否则，字典的后边界会向后移动 1 个字符，与之对应的，处理完一个长度为 n 的有匹配字符串后，字典的后边界会向后移动 n-4 个字符，如图 2-1（c）。字典的前后边界之间最大可容纳 64 千字节（Kilo Bytes，KB）的数据，并且，一旦数据量达到 64KB，字典的前边界将跟随后边界一起移动，以保证字典中的数据量始终是 64KB，如图 2-1（d）。

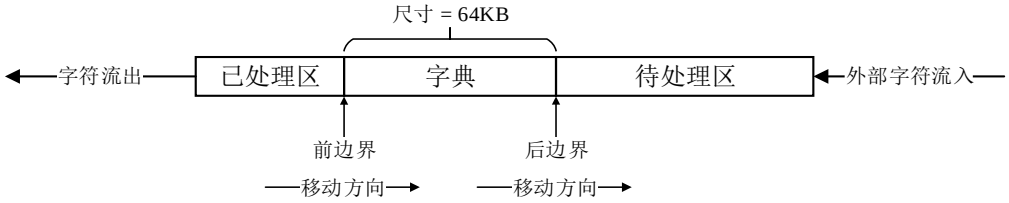
（a）初始状态的缓冲区



（b）读入首个四字符串的缓冲区



（c）字典未填满的缓冲区



（d）字典填满的缓冲区

图 2-1 压缩过程中字典和缓冲区状态

由于字典中的数据量比较大，如果在查字典的过程中将待处理字符串与字典中的每个字符串依次进行对比会耗费大量的时间，所以查字典的过程引入了基于 Hash 表的快速查找方法。其中，Hash 表是一种通过建立随机的键 *Key* 与值 *Value* 之间的映射关系来实现存储地址与存储数据之间耦合的数据结构[错误!未找到引用源。]。在 LZ4 压缩算法中使用的 Hash 表深度为 32KB，即用于寻址的 Hash 值位宽为 15bit，Hash 表每个可寻址单元内存储的数据是四字符串的 32bit 绝对首地址，对应的 Hash 键为位宽 32bit 的四字符串。并且，LZ4 为了将 32bit 的 Hash 键对应到 Hash 表的 15bit 地址空间，使用了黄金分割 Hash 函数，如式（2.1）所示，其中，2654435761 是数值范围 0 到 2^{32} 之间的黄金分割值，输入变量为 Hash 键 *Key*，计算过程保证所有中间变量都被限制在 0 到 2^{32} 之间，截取计算结果的高 15bit 作为 Hash 值 *Value*。

$$Value = \frac{(Key \times 2654435761) \bmod (2^{32})}{2^{17}} \tag{2.1}$$

使用 Hash 表在字典中查找可能存在的匹配字符串，可以将查字典的操作控制在几个时钟周期之内，大大提升了查字典的速率。然而，由于 LZ4 的 Hash 表使用 15bit Hash 值来映射 32bit Hash 键必然会导致两个不同的 Hash 键计算出相同的 Hash 值的情况，也就是 Hash 碰撞[错误!未找到引用源。]。由于在 LZ4 中使用的是单级 Hash 表，所以对于 Hash 碰撞的处理是直接替换。例如，用四字符串 str_2 作为 Hash 键，计算出 Hash 值并索引到对应的 Hash 表存储单元，取出的是四字符串 str_1 的首地址，而 str_1 与 str_2 不相等，则判定发生了 Hash 碰撞，str_1 是 str_2 的伪匹配字符串，直接用 str_2 的首地址覆盖对应的 Hash 表存储单元里 str_1 的首地址即完成了 Hash 碰撞处理。查字典操作的目的是快速找到字典中可能存在的与当前正在处理的四字符串完全一致的四字符串，如果确认找到了这样的四字符串，则判定为真匹配，并且，只有在真匹配的同时保证两相同字符串首地址之间的距离不超过 65535 个字节（64KB），才被认定为一次有效匹配，并进入下一步骤的操作。查字典操作包含大量的存储器访问，延迟很大，是整个 LZ4 算法在压缩速率方面的瓶颈，而使用硬件电路实现时，可以充分利用片内随机访问存储器（Random Access Memory，RAM）的

低延迟和高带宽特性，大大降低查字典操作耗费的时间。

匹配操作只在查字典并找到了相同的四字符串之后才会进行，匹配的目标是从已在字典中找到的四字符串处尝试继续向后扩展匹配，直到在待处理字符串与字典字符串之间找到不相同的字符为止。匹配操作依次对比字典字符串和待处理字符串的每个后续字符，一旦找到不同的字符，就终止匹配过程，并将之前已匹配的字符串末尾 5 个字符组成 2 个四字符串首地址，写入到 Hash 表的对应位置。同时，获得正在处理字符串在字典中可以找到的匹配串的最大长度，即匹配长度；以及正在处理字符串首地址与字典中匹配串的首地址之间的偏移量，即匹配距离。匹配操作的同时会更新 Hash 表，在扩展匹配时，丢弃了大量的潜在的四字符串，导致后续的查找匹配过程中，成功找到匹配的概率降低，或匹配距离较长，压缩率变差。

替换操作是真正实现压缩功能的步骤，利用匹配操作中获得的匹配长度、匹配距离来替换在字典中存在匹配的字符串，达到减少字符串所占体积的目的，对于在字典中没有找到匹配的字符串也要一并放入压缩编码流中，以保证信息的完整性。由于替换操作的输出长度不确定，所以必须对数据进行移位拼接操作，一定程度上影响了压缩速率，需要设计一种并行移位的数据拼接的电路，降低移位拼接过程的延迟。

2.1.2 LZ4 压缩的输出格式

LZ4 压缩算法提供了一种便于存储和解压缩的帧格式，用于存储经过替换操作的数据流，如图 2-2 所示。其中，帧头占 1 个 Byte，其高 4bit 代表无匹配字符串长度，简称无匹配长度，低 4bit 代表有匹配字符串长度，简称匹配长度。两个扩展段都是可选部分，分别用于表示无匹配长度超过 15 的部分，以及匹配长度超过 19 的部分，每个扩展 Byte 可表示的最大值是 255，扩展 Byte 的个数随着无匹配长度或匹配长度改变。

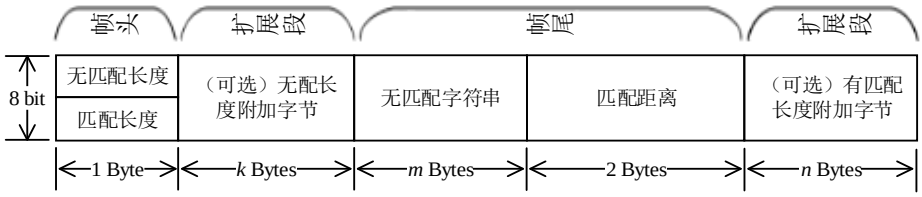


图 2-2 LZ4 压缩帧格式

对照图 2-2，假设一帧中的无匹配字符串长度为 i，有匹配字符串长度为 j，则无匹配长度附加 Byte 个数 k，有匹配长度附加 Byte 个数 n 的计算方法如式（2.2）所示。帧尾包括 m 个 Byte 的无匹配字符串和 2 个 Byte 的匹配距离构成，解压缩时扫描到帧尾就可以开始恢复原始数据了，先处理无匹配字符串，再处理有匹配字符串，直至完成一帧的解压缩。

$$k=\begin{cases}0 & i<15\\ 1 & i=15\\ \lceil (i-15)/255 \rceil & i>15\end{cases}\qquad n=\begin{cases}0 & j<19\\ 1 & j=19\\ \lceil (i-19)/255 \rceil & j>19\end{cases}\tag{2.2}$$

如图 2-3 所示，LZ4 压缩输出的数据流不仅是各个独立编码过程产生的数据帧的组合，而且数据流头

部包括用于识别数据流类型的 32bit 特征编号（Magic Number）和 32bit 数据流长度，尾部包含一个用 XXH32 快速摘要算法生成的 32bit 校验字，校验计算的对象是原始数据流。每次对 LZ4 格式的压缩数据流进行完整的解压缩后，便会计算解压结果的校验字，如果计算出的校验字与压缩数据流中的校验字一致，则认为压缩数据流中没有错误，已将原始数据完全恢复。由于 LZ4 输出数据流格式在各数据帧之间存在冗余，所以，LZ4 压缩算法在压缩率性能方面有一定的提升空间。

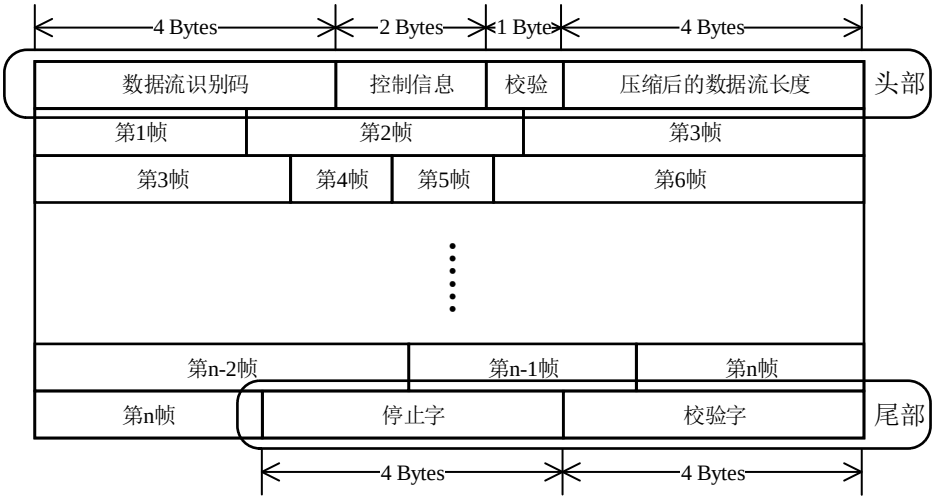


图 2-3 LZ4 输出数据流格式

2.1.3 校验算法

生成校验字的 XXH32 快速摘要算法的流程如图 2-4 所示，其分为复位、更新和摘要三个阶段。a_1、a_2、a_3 和 a_4 分别代表将输入 32bit 数据划分成的 4 个字节，PRIME_1 到 PRIME_5 代表 XXH32 算法包含的五个常系数，v_1、v_2、v_3 和 v_4 分别代表 XXH32 计算过程中的 4 个中间变量字节。式（2.3）代表的 ROTL 操作旨在把 32bit 数据整体循环左移规定数量的二进制位。复位阶段将 v_1 到 v_4 分别初始化为不同的常数。更新阶段依次计算 v_1 到 v_4，每计算一轮需要 16Byte 数据，计算方法如式（2.4）所示，其中 $v_i(r)$ 代表第 r 轮计算产生的 v_i。摘要阶段相对复杂，首先，获取待校验数据的字节数 total_len，并将校验值 h 初始化为 0；然后根据 total_len 值确定进入的计算分支；最后，如果剩余字节数少于 4，而又不为 0，则开始按字节进行处理，直到处理完所有字节；处理完成后还要经过一系列移位异或和累乘操作才能获得最终的 XXH32 校验值 h。

$$\mathrm{ROTL}(x,i)=(x<<i)\,|\,(x<<(32-i)) \tag{2.3}$$

$$v_i(r)=\mathrm{PRIME\_1}\times\mathrm{ROTL}\left((v_i(r-1)+a_i(r)\times\mathrm{PRIME\_2}),13\right) \tag{2.4}$$

校验算法的流程复杂，步骤繁多，并且包含大量串行的乘法和移位操作，导致压缩速率降低。使用硬件电路的方法实现校验算法可以极大的提高移位和乘法运算的效率，减小处理延迟。

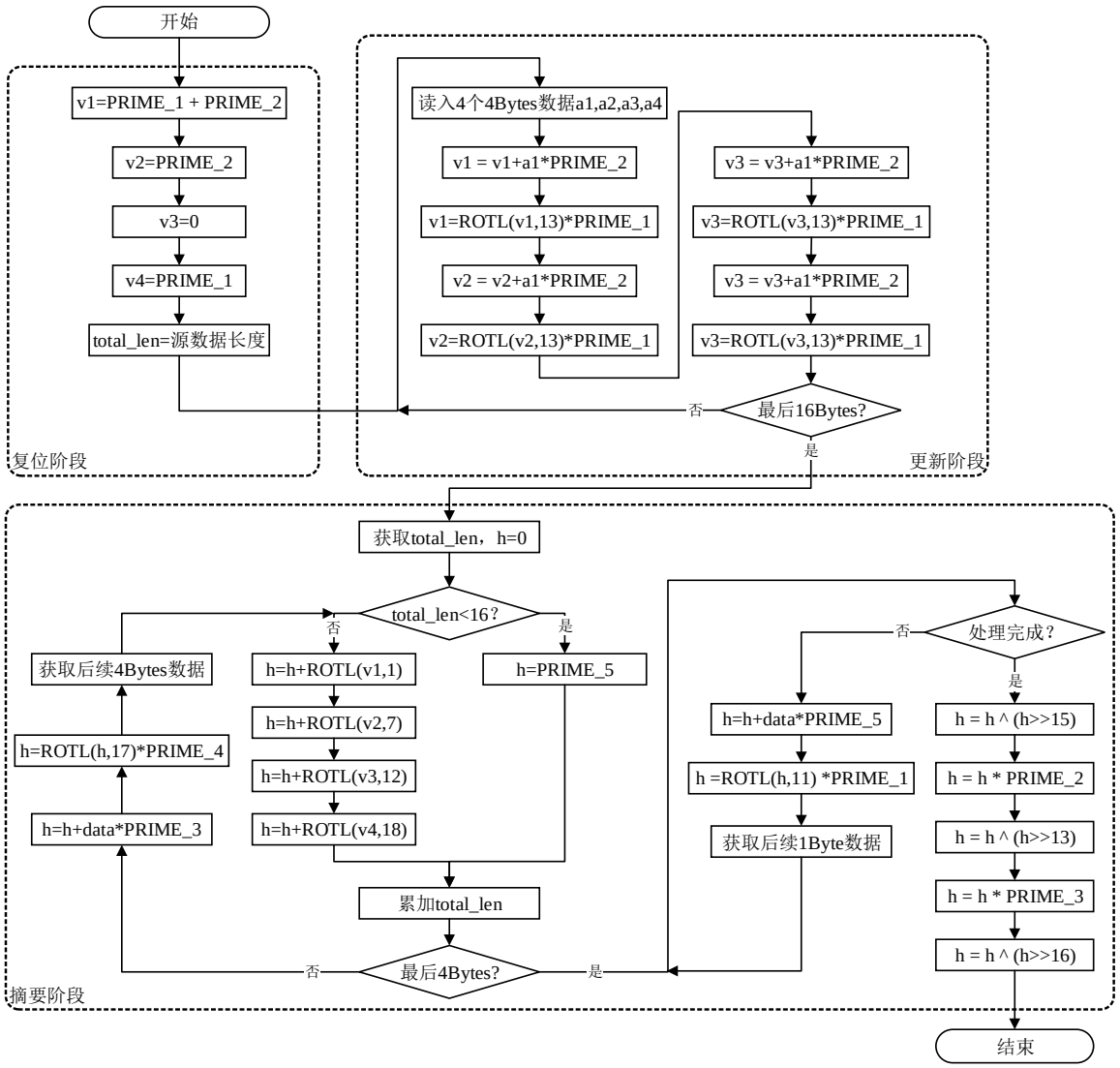


图 2-4 XXH32 快速摘要算法流程图

综上所述，一次完整的 LZ4 压缩过程包括多次匹配和替换操作、校验字生成操作、以及所有 LZ4 帧的组织和拼接操作。其中最关键的是匹配和替换操作，每次匹配的命中概率、匹配长度、匹配距离等参数直接决定了压缩率性能。

2.2 Huffman 编码算法分析

2.2.1 Huffman 编码的特性

Huffman 编码是由 David A. Huffman 提出的一种基于统计的变长编码（Variable Length Code，VLC）方法[错误!未找到引用源。]。经过多年的研究实践，Huffman 编码已被证明是去冗余效果最好，平均编码长度最接近信源熵的变长编码方法[错误!未找到引用源。]。式（2.5）代表平均编码长度，其中 N 为当前样本中所有出现过的字符总数，将这些字符从 1 到 N 进行编号，P_i 是编号为 i 的字符出现的概率，l_i 是编号为 i 的字符变长编码长度，L 则是对当前样本进行变长编码的平均码长。

$$L=\sum_{i=1}^{N}P_i\times l_i \tag{2.5}$$

式（2.6）代表信源熵的定义，其中 N 为样本中出现过的字符总数，P_i是编号为 i 的字符出现的概率，由于 $0<P_i\le 1$，所以 $\log_2 P_i<0$，求和之后取相反数就是通过现有字符样本统计得到的信源熵 H，其值代表变长码压缩所能获得的最短平均码长。

$$H=-\sum_{i=1}^{N}P_i\times \log_2 P_i \tag{2.6}$$

使用变长编码对样本数据进行压缩时，有 $L\ge H$，当且仅当使用 Huffman 编码对样本数据进行压缩时才有 $L=H$，这种性质称为 Huffman 编码的最优性[错误!未找到引用源。]。

由于 Huffman 编码是一种变长编码，其输入定长数据，输出变长数据，使输出结果的平均码长小于原始定长码长度，达到压缩的目的。实现变长码的难点是如何保证其独特可译性，即不同字符的变长码之间不能存在连续重叠的情况，否则将导致编码后的数据不能被正确解码，使被压缩的数据丢失。使用 Sardinas-Patterson 定理来判断一组字符对应的变长码集合是否具备独特可译性[错误!未找到引用源。]。判别方法可描述为，假设 $\exists C_m,C_n\in C_{set}$，当 C_m 是 C_n 的前缀，而 C_n 剩余的后缀 $C_k\in C_{set}$，则可判定该变长码集合 C_{set} 是非独特可译的，否则，该集合为独特可译码集合。基于二叉树的 Huffman 编码一定是独特可译码[错误!未找到引用源。]。

2.2.2 Huffman 编码的分类

现有的 Huffman 编码主要分为静态和动态两种编码方法。静态编码是一种查表编码方法，即在编码之前就已获得了信源字符的概率分布，并且生成了对应的变长编码表，编码时直接查表替换即可。静态编码被广泛的应用于有损的音视频压缩，例如 MP3 标准[错误!未找到引用源。] 和 H.264 标准[错误!未找到引用源。]；部分无损压缩，例如 Deflate 标准[错误!未找到引用源。]。动态编码是一种在编码过程中对全部或分块的待压缩数据进行统计，并用统计结果预测信源字符概率分布并生成变长编码表进行编码的方法，常用于无损压缩[错误!未找到引用源。]。由于静态编码预先设置了变长编码表，所以其压缩速率很高，然而一旦预设的码表与当前处理的数据块中字符概率分布差异较大时，其平均码长会明显大于数据块的熵值，导致压缩率变得很差。相反，由于动态编码能不断进行统计和更新码表，其平均码长始终能够接近或等于数据块的熵值，压缩率能保持在比较好的水平，然而其分块统计和建立二叉树的过程消耗了大量的时间，压缩速率远低于静态编码。

为了追求压缩速率，大多数无损压缩电路中使用了静态编码方法[错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。错误!未找到引用源。]，导致了压缩电路的压缩率普遍比压缩软件差。本文使用的编码方法是基于一种介于动态编码和静态编码之间，被称之为半静态 Huffman 编码的思想[错误!未找到引用源。]。半静态 Huffman 编码吸取动态编码压缩率性能较高，以及静态编码压缩速率较高的优点，其初始化统计过程与动态编码一致，编码过程与静态编码一致。

2.2.3 动态 Huffman 编码流程

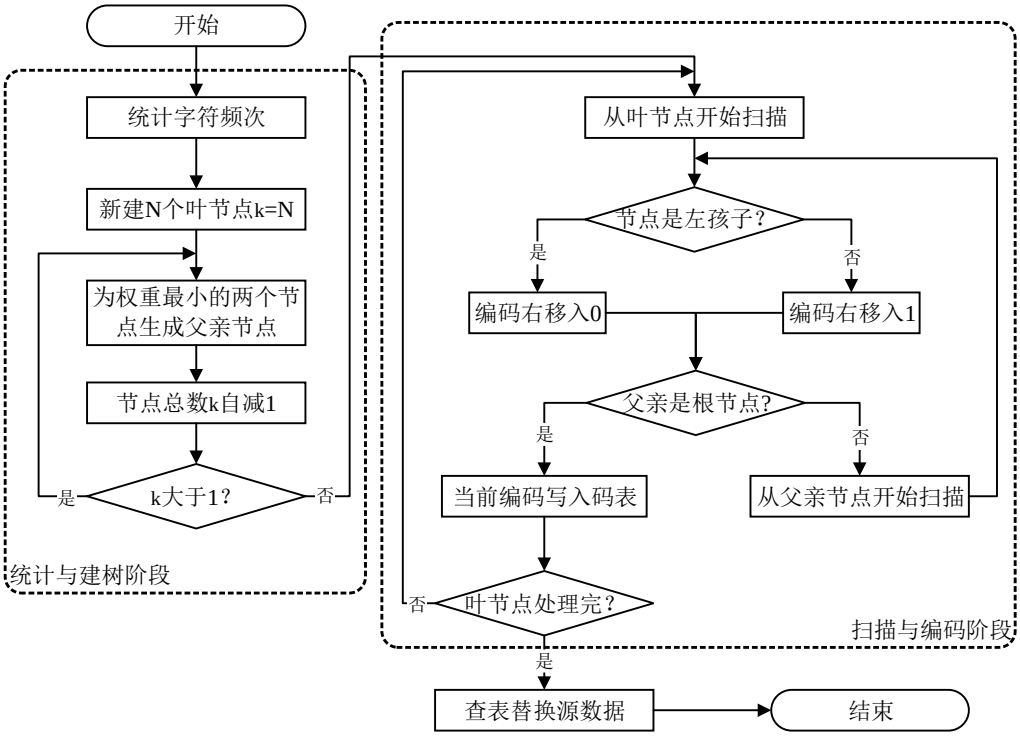


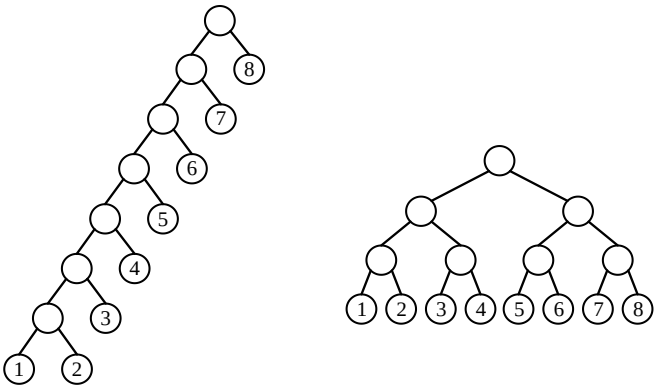
图 2-5 动态 Huffman 编码流程图

动态 Huffman 编码的基本思想是建立一个以所有出现过的字符为叶节点的二叉树，然后把从叶节点到根节点的路径转化为变长编码。其编码流程如图 2-5 所示，首先统计待压缩数据的各字符出现次数；统计结束后，发现共有 N 个字符出现的次数不为 0，于是新建 N 个对应这些字符的二叉树叶节点，编号 i 的范围从 1 到 N，每个叶节点对应字符出现的次数作为叶节点的权重，记作 Freq[i]；从 N 个叶节点中选出权重最小的编号为 x 和 y 的两个节点，为它们生成新的父亲节点，也称为中间节点，其权重是 Freq[x]与 Freq[y]之和，此时未处理的叶节点和生成的中间节点总数为 k=N-1；然后从上一步得到的 k 个节点中再取出两个权重最小的节点，合并成为新的节点，节点总数减 1；重复上一步操作，直到节点总数 k 等于 1，此时剩余的节点称为根节点，是二叉树最上层的节点；最终，二叉树拥有了 N 个叶节点、N-2 个中间节点和 1 个根节点，以每个叶节点作为起始向根节点逐位进行编码，如果当前经过的节点是其父亲节点的左孩子节点，则把编码寄存器右移入一个 0，如果是右孩子节点则右移入一个 1；如果当前节点的父亲是根节点，则把编码寄存器值写入变长编码表，清除编码寄存器，开始扫描下一个叶节点；重复扫描过程，直到所有叶节点都生成了对应的编码，变长编码表已建立，即可以把待压缩数据中的字符用变长编码逐个替换，从而降低平均码长，达到压缩的目的。

2.2.4 动态 Huffman 编码的缺陷和优化方法

对于一种通用的无损压缩算法而言，由于信源模型的不确定性，待压缩字符出现次数的概率分布难以估计，导致依此建立的二叉树深度范围很大。假设一共统计到了 8 个出现次数不为 0 的字符，分别对应编

号为 1 到 8 的八个叶节点，建立的二叉树可能出现图 2-6 中的两种极端情况。其中，图 2-6（a）所示的偏置二叉树拥有 8 个叶节点，从最上方的根节点开始作为树的第 0 层、节点 8 所在层为第 1 层，以此类推，树共有 7 层，深度为 7；而图 2-6（b）所示的满二叉树同样拥有 8 个叶节点，深度仅为 3。这就意味着对 8 个字符建立的二叉树深度从 3 到 7 不等，而通过扫描树得到的变长编码的码长范围也在 3 到 7 之间。扩展到 256 个字符的情况，建立的二叉树深度范围从 8 到 255 不等。尽管生成的二叉树准确的反映了当前待压缩数据中字符的分布情况，但在对树进行扫描并生成变长编码表的过程中，二叉树的深度越深，扫描越慢，且位宽固定的编码占用的存储空间也越多。所以，无论是从时间复杂度还是空间复杂度而言，生成的二叉树的深度并不是越大越好。

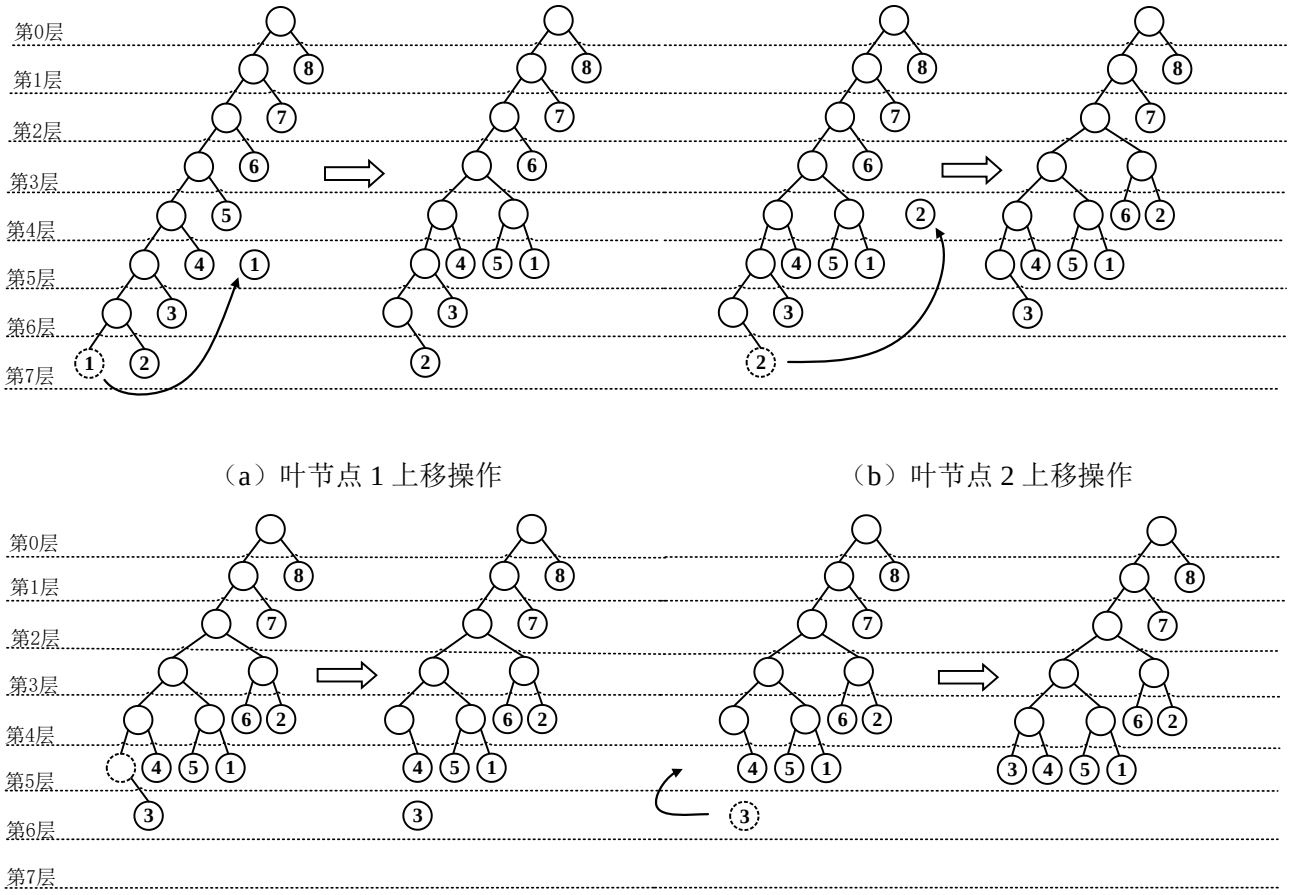


（a）偏置二叉树　　（b）满二叉树

图 2-6 两种拥有 8 个叶节点的二叉树

尽管基于统计的动态 Huffman 编码具有唯一性[错误!未找到引用源。]，即针对确定概率分布的字符集合，生成的二叉树及其变长编码表是唯一的。但是，为了避免出现如图 2-6（a）所示的偏置二叉树的情况，在统计与建树阶段可以对生成的二叉树进行树优化，将超出规定深度的叶节点移到其它子二叉树的分支上，减小树的深度。以图 2-6（a）中的情况为例，假设人为的规定二叉树最大深度为 5，最大码长为 5，则节点 1、2、3 均属于超出规定深度的叶节点，它们代表了三个不同的字符，称为溢出字符，即编码长度溢出最大码长的字符。如图 2-7（a）所示，首先将节点 1 移到第 5 层，由于该二叉树没有空闲的中间节点，如果要把节点 1 放在第 5 层，需要在第 4 层新建一个空闲的中间节点作为其父亲，这就需要把第 4 层的节点 5 往下移一层，从而释放一个孩子节点的位置，供新建的中间节点使用，然后把新建的中间节点作为节点 1 和节点 5 的父亲，节点 1 就被成功的从溢出的第 7 层移到了第 5 层。然后处理节点 2，如图 2-7（b）所示，由于此时第 4 层已经没有叶节点可供释放孩子节点的位置，所以第 5 层暂时无法插入节点，只能将节点 2 插入第 4 层，并将第 3 层的节点 6 移到第 4 层，建立新的中间节点作为节点 2 和 6 的父亲，并且，原先节点 1 和 2 在第 6 层的父亲节点已经没有孩子节点，可以直接删除。最后处理节点 3，如图 2-7（c）所示，由于节点 3 是其父亲节点的唯一孩子节点，所以可以删除其父亲节点，并用节点 3 替换它，如图 2-7（d）所示。经过优化后，所有的溢出字符叶节点均已被放入深度限制为 5 的二叉树中，可以被扫描并获得码长在 5 以

内的变长编码表。



（a）叶节点 1 上移操作　　（b）叶节点 2 上移操作

（c）中间节点删除操作　　（d）叶节点 3 上移操作

图 2-7 树优化过程示例

需要注意的是，上述的树优化过程改变了部分叶节点在二叉树中的位置，也间接改变了部分叶节点的权重，使优化后的树偏离了待压缩数据的统计特征，将必然导致优化后的动态编码的压缩率变的稍差一些。优化算法本身也充分考虑了这一情况，由于越靠近根节点的叶节点编码长度越短，在待压缩数据中出现的次数也越多，如果该叶节点被下移一层，增加的平均码长也越多，所以优先将溢出叶节点插入到二叉树最深的那一层，以减小对上面几层的叶节点的影响。综合考虑算法实现需要的运行时间和空间占用，使用优化的二叉树可以更好的权衡硬件资源消耗、压缩速率与压缩率。

除了在统计与建树阶段可以对算法的实现过程进行优化外，在扫描与编码阶段也可以使用一些改进的方法来实现相同的功能。传统的通过扫描树进行 Huffman 编码的方法无论使用软件还是硬件实现都非常耗时，如何快速的实现编码过程是提升动态 Huffman 编码速率的关键。考虑到二叉树的父亲节点权重大于它的任意一个孩子节点的权重，而两个孩子节点之间并没有明确的权重大小关系，也就是左右孩子节点可以互换。基于这样的特性，可以通过给定不同码长之间不重叠的计数初值，计数生成相同码长的不同字符的编码，快速确定每个字符各自对应的独特可译变长编码。该方法保证变长编码独特可译性的基本原理是利用生成码长较长的计数初值前缀不会出现在码长比它更短的变长编码子集中。通过给定不重叠的计数初值

实现计数生成变长编码的方法如表 2-1 所示，其中k为当前树的最大深度，第 1 层是码长最短的节点，其计数初值从 0 开始，到n_1-1结束，共n_1个字符被编码；第 2 层的计数初值为$2n_1$，其前缀为n_1，不同于第 1 层的任何一个变长编码码字；第 3 层的计数初值为$2(2n_1+n_2)$，其前缀为$2n_1+n_2$，不同于第 2 层或第 1 层中的任何一个码字；以此类推，直到第k层编码完成后，依然可以保证整个变长编码集合的独特可译性。

表 2-1 计数生成变长编码的方法

层编号	编码个数	计数初值	计数终值
1	n_1	0	n_1-1
2	n_2	$2n_1$	$2n_1+n_2-1$
3	n_3	$4n_1+2n_2$	$4n_1+2n_2+n_3-1$
4	n_4	$2^3n_1+2^2n_2+2n_3$	$2^3n_1+2^2n_2+2n_3+n_4-1$
……	……	……	……
k	n_k	$\sum_{i=1}^{k-1}2^{k-i}n_i$	$\left(\sum_{i=1}^{k-1}2^{k-i}n_i\right)-1$

不仅如此，表 2-1 中所示的生成变长编码的方法还保证不增加额外的码长，其原因是如果在第k层的计数终值达到2^k-1，则第k层将不存在中间节点，树已到达最深的那一层。如式（2.7）所示，假设第k层包含中间节点和叶节点，则有叶节点数n_k的不等式关系。

$$n_k \le 2^k-\sum_{i=1}^{k-1}2^{k-i}n_i \tag{2.7}$$

式（2.7）转化为式（2.8）所示的形式。

$$n_k+\sum_{i=1}^{k-1}2^{k-i}n_i \le 2^k \tag{2.8}$$

即式（2.9）。

$$\sum_{i=1}^{k}2^{k-i}n_i \le 2^k \tag{2.9}$$

当且仅当第k层不存在中间节点时，式（2.9）中才可取等号。

2.3 本章小结

本章主要描述了 LZ4 压缩算法的基本原理，对其中的关键步骤的细节进行了分析。并且，描述了 LZ4 压缩输出数据流格式，以及压缩附加信息中的校验算法。描述了变长编码的独特可译性原理，并分析了静态和动态 Huffman 编码的优缺点。着重考察了动态 Huffman 编码在建树过程中的缺陷，并提出针对性的优化方法。

第三章 LZ4 算法的优化及压缩电路设计

第三章针对原始 LZ4 压缩算法中为了保证压缩速率而牺牲压缩率的部分进行了优化，利用硬件电路并行化的特点，提出了一种在不降低压缩速率的前提下提升压缩率的方法，并实现了相应的 LZ4 无损压缩电路。

3.1 LZ4 无损压缩算法优化

一次完整的匹配过程包括查找匹配和扩展匹配两个阶段，如图 3-1 所示，其中查找匹配阶段用于确定匹配开始的位置，扩展匹配阶段用于确定最大的匹配长度。受制于通用计算机系统的串行处理架构，原始的 LZ4 压缩算法为了保证压缩速率，在扩展匹配阶段忽略了大量的可匹配字符串，这些字符串对应的地址或位置信息不会被录入 Hash 表中，以致于在后续的查找匹配过程中不能通过 Hash 表被索引到，从而导致一些潜在的可匹配字符串被忽略，使压缩率性能变差。如图 3-2 所示，假设匹配头及其后方的下划线字符串是可以完整匹配的字符串，各子字符串对应的地址使用符号&表示，在 LZ4 压缩软件处理过程中，只将匹配头 4 字节和匹配尾 5 字节共 3 个字符串地址录入 Hash 表，匹配头和匹配尾之间的字符串都被忽略了。

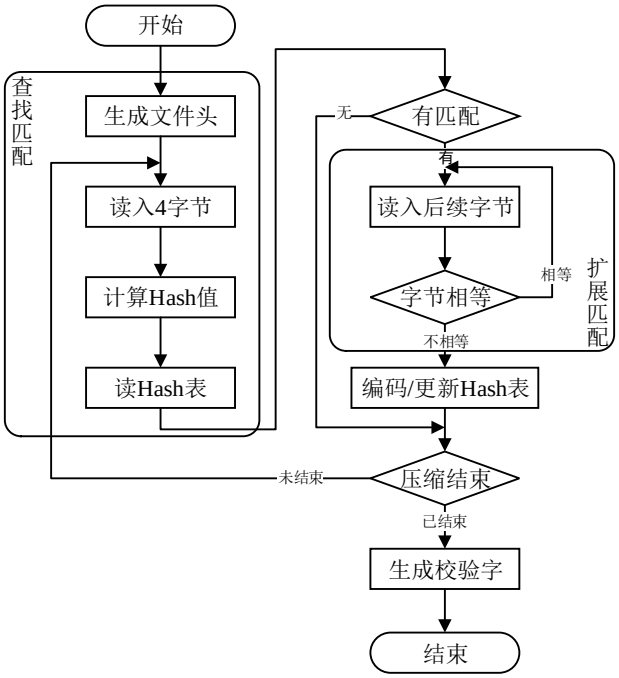


图 3-1 匹配过程流程图

使用专用硬件电路实现则可以将扩展匹配过程和 Hash 表录入过程并发处理，即在扩展匹配的同时计算各字符串的 Hash 值，并将各字符串地址放入到 Hash 表中对应的存储单元内，提高了 Hash 表的更新频次。由于扩展匹配时进行对比的单位是 4 字符串，所以每个时钟周期需要录入 Hash 表的字符串地址是 4 个。如图 3-3 所示，Hash 表 1 和 Hash 表 2 是同属于一个 Hash 表的不同区段。该示例中，扩展匹配过程录

入 Hash 表的字符串地址由图 3-3 中的 3 个增加为图 3-4 中的 13 个，使后续字符串找到匹配的概率获得提升。极端情况下，频繁的更新 Hash 表会导致字典中匹配距离较大，且具有更多可扩展匹配的字符串的地址被匹配距离较小，且难以扩展匹配的字符串地址所取代，可能导致压缩率变差。上述改进方法如果使用软件实现势必显著降低压缩速率，使 LZ4 压缩算法的速度优势被削弱，所以，该压缩率优化方法只适用于硬件压缩电路实现。

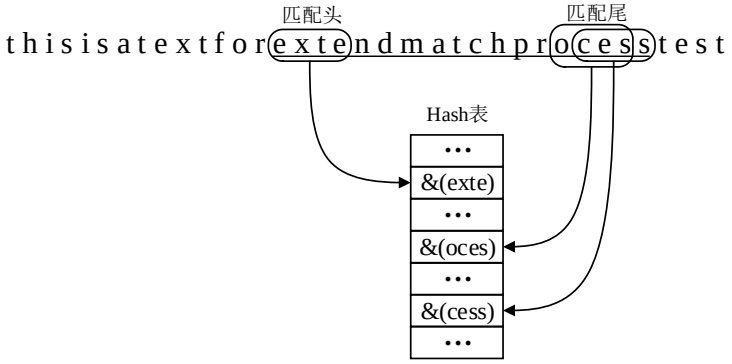


图 3-2 优化前的扩展匹配 Hash 表更新过程示例

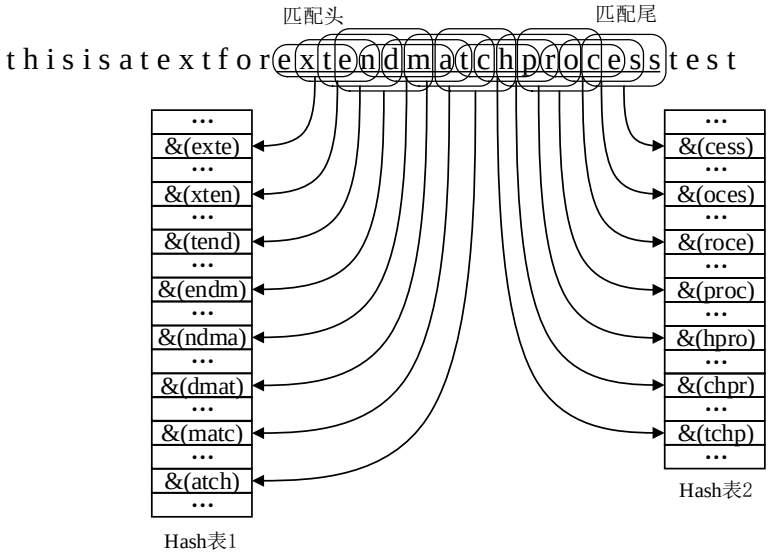


图 3-3 优化后的扩展匹配 Hash 表更新过程示例

由于 LZ4 算法的输出数据还可以被二次压缩，能够进一步优化压缩率。故本文使用半静态 Huffman 编码进一步压缩 LZ4 算法的输出结果，获得比提高 Hash 表更新频次的方法更好的优化效果，关于半静态 Huffman 编码算法及其电路设计将在第四章详细阐述。

3.2 LZ4 压缩电路设计

根据 2.1 节所述的 LZ4 压缩基本原理和 3.1 节所述的压缩率优化方法，设计的一种优化的 LZ4 压缩电路，其中包括字典缓冲器、匹配电路、字符串分割电路、并行编码器、校验电路和流水线控制器。

3.2.1 总体架构

LZ4 压缩电路的总体架构如图 3-4 所示，包含字典缓冲器、分割电路、匹配电路、编码电路和校验电路共五个功能性模块电路，以及一个用于协调各模块电路工作和数据交换的流水线控制器。

字典缓冲器用于从外部接收并缓存待压缩数据，并根据流水线控制器提供的控制信号确定字典的前边界和后边界在已缓存数据中的位置，字典前边界用于分隔有效字典与字典以外的已处理数据，字典后边界用于分隔有效字典与待处理数据。字典缓冲器向后级电路提供可顺序访问 FIFO 接口和可随机访问的 RAM 接口，以分别满足向后级输出待压缩字符串和被后级随机查字典的需求。

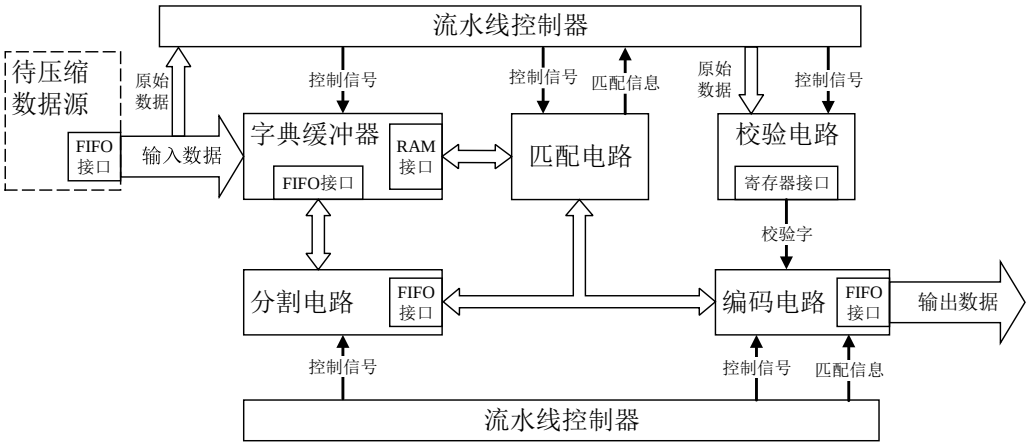


图 3-4 LZ4 压缩电路的总体架构

由于输入数据流是 4Byte 宽度的，而 LZ4 压缩处理的基本单位是 1Byte，其字典后边界有可能位于 4Byte 数据的中间位置。于是，使用分割电路将待压缩 4Byte 数据进行按字节移位和拼接，使每次进入匹配电路的数据都是紧邻字典后边界且未被处理的四字符串。因为待处理字符串必须按顺序的交给后级电路，所以分割电路仅提供 FIFO 接口供后级使用。

通过 RAM 接口，匹配电路在字典缓冲器中查找来自分割电路的待压缩四字符串，如果在 Hash 表中找到有效的匹配字符串，则将匹配距离、匹配长度、无匹配长度等匹配信息交给流水线控制器，最终转移给编码电路。否则，更新 Hash 表，并开始下一次查找过程。

校验电路从流水线控制器接收未压缩的原始数据，使用 XXH32 快速摘要算法生成校验字，用于 LZ4 压缩数据流在解压缩过程中的正确性检查。由于校验字只有 32bit，所以只需要向后级电路提供寄存器接口。

编码电路用于将匹配距离、匹配长度、无匹配长度和无匹配字符串这四个信息进行组合，编码成符合 LZ4 格式的压缩帧。然后加入原始数据信息和校验电路的输出数据，形成 LZ4 输出数据流。输出数据以 FIFO 接口的形式顺序输出。

流水线控制器用于协调控制各模块电路的工作，并负责在各模块电路之间转移关键信息和数据。

3.2.2 字典缓冲器

图 3-4 中的字典缓冲器用于暂存数据，并将整个缓冲器分为字典、已处理区和待处理区。其内部包含一个 32bit 位宽，32K 深度的双端口 RAM 以及对应的冲突处理逻辑。以及一个负责协调外部的 FIFO 写、FIFO 读、RAM 读、RAM 写四种请求，并同时产生 RAM 访问地址的电路，称为仲裁/地址逻辑。字典缓冲器中通过分时复用的方法实现将双端口 RAM 虚拟出四个相互独立的接口，包含一个 FIFO 写接口，一个 FIFO 读接口，一个 RAM 读接口和一个 RAM 写接口。

如图 3-5 所示，FIFO 写接口主要用于从外部接收待压缩的数据；FIFO 读接口主要用于向分割电路输出缓冲后的待压缩数据；RAM 读接口给匹配电路提供随机寻址的通道，使匹配电路可以从字典的任意位置取出字符串进行对比；RAM 写接口用于从冲突处理逻辑中提取出可能存在多个请求产生冲突时对应的地址和数据。

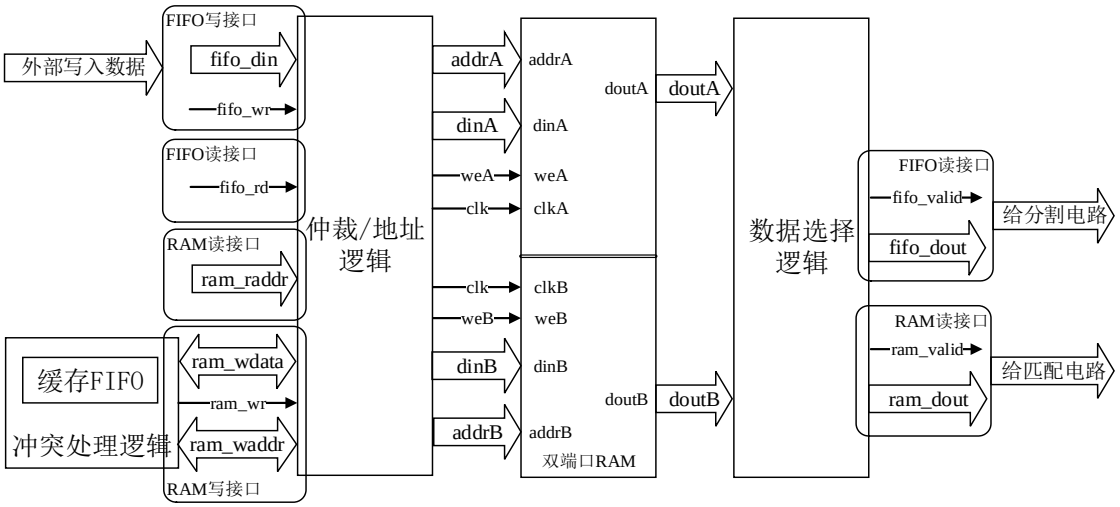


图 3-5 字典缓冲器电路框图

冲突处理逻辑中包含一个 47bit 位宽，512 深度的缓存 FIFO，在有 2 个以上的存储器访问请求同时发生时，冲突处理逻辑将其中优先级较低的请求对应的 15bit 地址和 32bit 数据存入缓存 FIFO 中，直到仲裁/地址逻辑在一个时钟周期内只接收到一个存储器访问请求时，才会响应之前被暂存的请求。数据选择逻辑用于分配 RAM 的两个输出端口数据给对应的后级电路。

3.2.3 分割电路

在图 3-4 中，分割电路用于接收来自字典缓冲器的待压缩数据，并且在查找匹配阶段对移位寄存器进行按字节移位，在扩展匹配阶段按指定字节数进行移位，从而实现匹配过程中字符串按字节分割的功能。如图 3-6 所示，分割电路中包含一个 12 字节并行移位寄存器组，移位方向从左到右，每次移动的字节数受控于移位控制逻辑，其中字节 B9 到 B12 用于向外输出四字符串，滑动指针指向寄存器组中的任意四个连续字节，并写入预取逻辑从外部获取的 4 字节输入数据。指针控制逻辑根据移位寄存器组中存储有效数据

的数量，来控制滑动指针指向的位置，使新字符串被写入到移位寄存器组中旧字符串的末尾，保证寄存器组中数据的连续性。

分割电路中用于从外部读入数据的预取逻辑状态转移过程如图 3-7 所示，其中 INIT 状态用于从字典缓冲器中读取 3 个 32bit 数据并初始化移位寄存器；如果在初始化阶段外部 FIFO 被读空，则转移到 WAIT_INIT 等待 FIFO 再次不为空；外部信号 rd_data_en 有效且 FIFO 不为空时，转移到 LOAD_SHIFT 对移位寄存器进行移位并从 FIFO 读取新数据将移位寄存器填满，如果上一个读出的数据还没有被写入移位寄存器，则不再读入新数据，而如果上一个新数据不存在，则并转移到 LOAD_ONLY 从 FIFO 读取一个数据；在 rd_data_en 无效时，转入 WAIT_SHIFT 等待；LOAD_SHIFT 和 LOAD_ONLY 状态下，一旦 FIFO 为空，则转移到 WAIT_FIFO 状态等待后续数据。

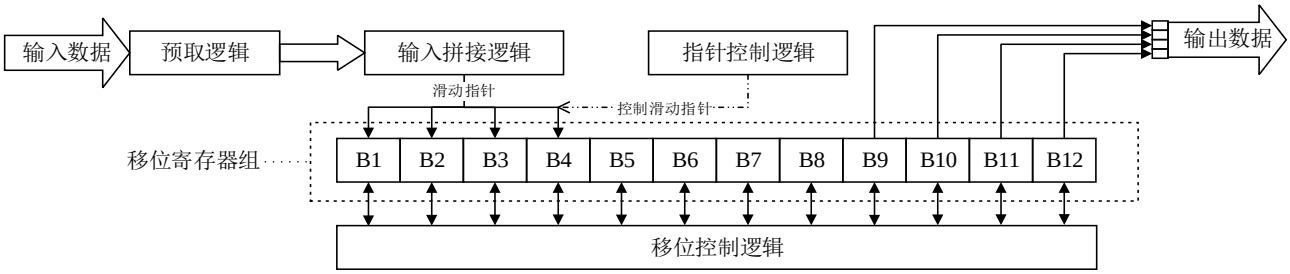


图 3-6 分割电路结构

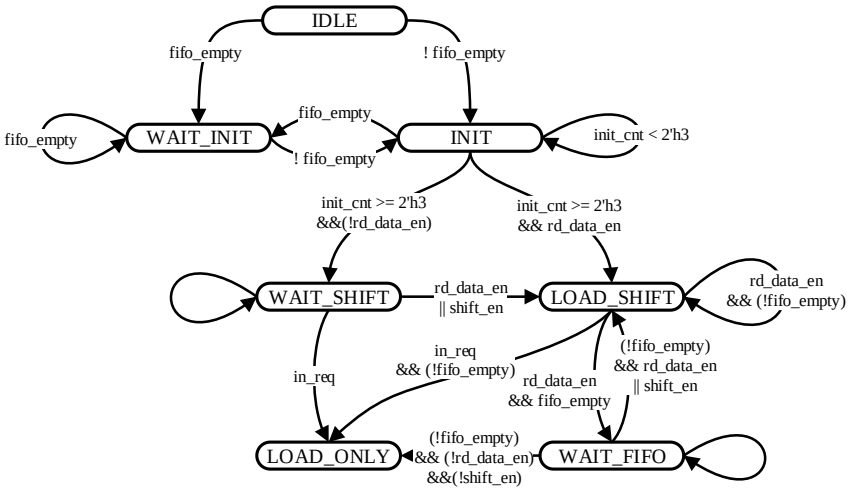


图 3-7 分割电路状态转移图

$$\begin{cases}(remind-shift\le 4)\ \text{and}\ (data\_valid=1)\\(remind-shift\le 8)\ \text{and}\ (data\_valid=0)\end{cases} \tag{3.1}$$

分割电路工作状态的转移由当前移位寄存器中剩余字节数 remind、外部请求移位的字节数 shift，当式（3.1）中任意一条成立时，就从外部 FIFO 接口读入一个 32bit 数据送入移位寄存器。其中 data_valid 为 1 代表从外部 FIFO 接口读入的数据已到达移位寄存器，可以在下一时钟周期被写入移位寄存器；为 0 代表当前无数据到达移位寄存器。

3.2.4 匹配电路

如图 3-4 所示，匹配电路负责从分割电路取出待匹配字符串，并通过 Hash 表索引字典中可能存在的匹配串并进行对比，最终向编码流水线控制器输出匹配长度、匹配距离、无匹配长度、无匹配字符串信息，用于后续编码。匹配电路中包含 4 路并行硬件乘法器（MUL），可以在 1 个时钟周期内完成两个 32bit 二进制无符号数的乘法运算；由四端口 RAM 构成的 Hash 表存储器，深度为 32768，位宽为 64bit，其中高 32bit 记录字典字符串的绝对首地址，低 32bit 记录字典字符串的首个 4 字符串；移位匹配逻辑，利用 Hash 表寻址字典缓冲器，查找可能存在的匹配字符串；并行匹配逻辑，用于在扩展匹配过程中进行 4 字符串的对比，并获得匹配串的长度和下一次匹配的起始位置。

匹配电路的框图如图 3-8 所示，其中字节比较器用于比较单个字节，双字比较器用于同时比较 4 个字节。Hash 表存储器的端口 A 被并行匹配逻辑和移位匹配逻辑分时复用，端口 B、C、D 被并行匹配逻辑独占，并行匹配逻辑和移位匹配逻辑通过外部 FIFO 接口从上一级分割电路取出待匹配字符串，通过 RAM 接口从字典缓冲器取出字典中的匹配字符串。并行匹配逻辑的四路输出 dina、dinb、dinc、dind 分别是当前正在处理的 4 字符串与前一个 4 字符串构成的四个 4 字符串。

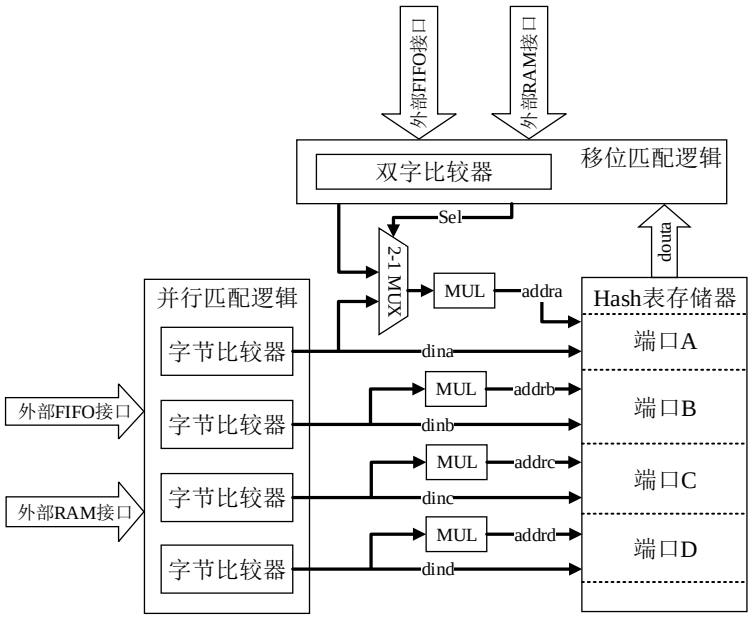


图 3-8 匹配电路框图

例如，图 3-9 中下划线字符串为当前正在处理的 4 字符串，str1、str2、str3、str4 是与上一个 4 字符串拼接生成的四个字符串，被送入四个固定系数乘法器，计算出 Hash 表地址 addra、addrb、addrc、addrd，并将四个字符串在数据流中的地址或位置信息 dina、dinb、dinc、dind 分别写入到对应上述 Hash 表地址的存储单元中。

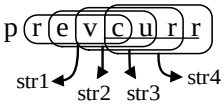


图 3-9 生成四个字符串示例

3.2.5 编码电路

编码电路用于生成图 2-2 和图 2-3 描述的 LZ4 格式编码流。编码电路包含拼接逻辑和字符 FIFO。编码电路对一个数据流进行一次完整的编码流程如图 3-10 所示。

在头部编码阶段，编码电路将预先设置的识别号（magic_number）、原始数据块尺寸（block_size）、以及压缩参数信息（info）按顺序拼接并写入输出流缓冲器。

在各帧编码阶段，编码电路通过 FIFO 接口从上一级匹配电路获取匹配长度（match_len）、匹配距离（match_dist）、无匹配长度（unmatch_len）、以及暂存在字符 FIFO 中的无匹配串字符（unmatch_lit），并通过拼接逻辑实现将上述四种匹配信息组合成压缩数据帧，多次重复这一过程，直到所有匹配信息都已处理完成。

编码电路在一次完整的数据流编码过程最后，从校验电路的寄存器接口读取 32bit 校验字，放置在压缩数据流的尾部，即尾部编码阶段。

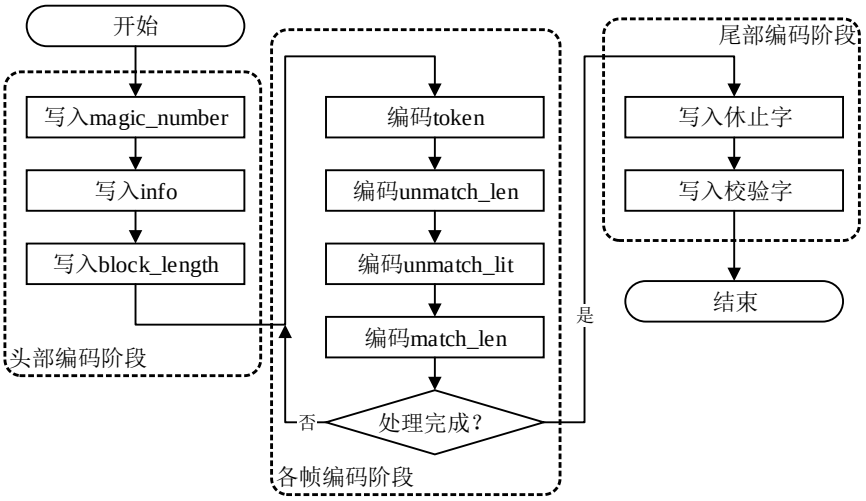


图 3-10 编码电路工作流程图

实现 LZ4 编码的关键电路是拼接逻辑，如图 3-11 所示，其中包含一个累加计数器、一个 64bit 移位寄存器和一个位宽为 32bit 的输出 FIFO。外部输入的待拼接数据统一为 32bit 高位对齐形式；与数据同时输入的还有字节计数，用于指明当前 32bit 数据中有效的字节数。移位寄存器用于实现数据拼接和按字节移位，累加计数器用于计算当前移位寄存器中的有效字节数，并在移位寄存器中有效字节数超过 4 时，向输出 FIFO 发送进位信号，将高 4 字节存入输出 FIFO 中，便于后面的半静态 Huffman 编码电路读取。

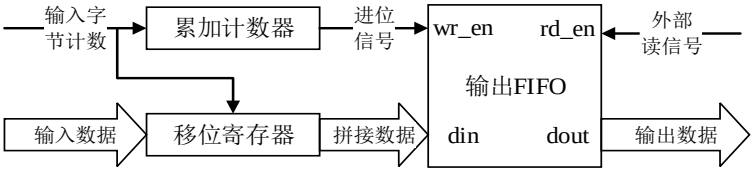


图 3-11 拼接逻辑框图

3.2.6 校验电路

检验电路用于生成图 2-3 中，位于 LZ4 输出数据流尾部的 32bit 校验字。

如图 3-12 所示，校验电路包含缓冲器、算法控制器、3 个乘法器（MUL）和 3 个移位寄存器（SHIFTER），算法控制器并行使用 3 个 MUL 实现乘法操作，并行使用 3 个 SHIFTER 实现移位操作，根据图 2-3 中的 XXH32 快速摘要算法流程图生成 32bit 校验字，交给寄存器接口，供后级电路使用。

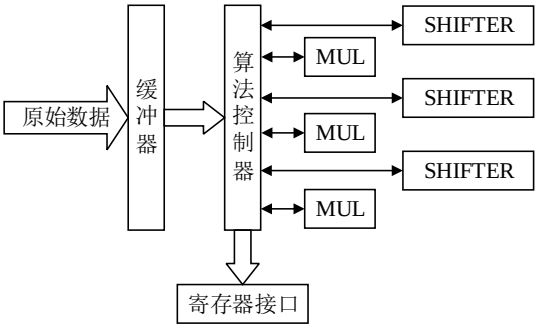


图 3-12 校验电路框图

校验电路的状态转移过程如图 3-13 所示。

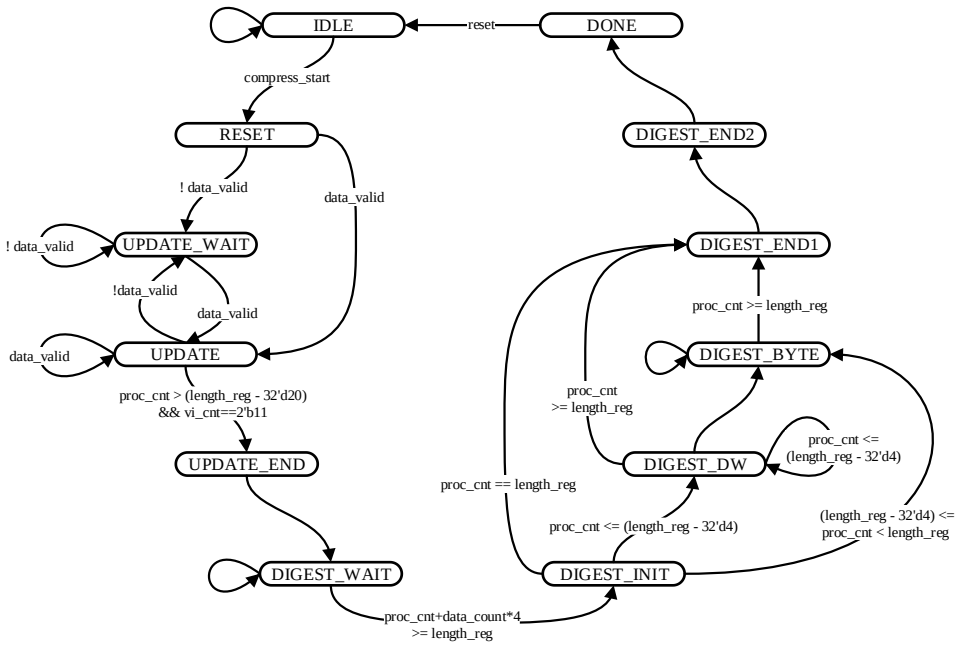


图 3-13 校验电路状态转移图

电路接收到压缩开始 compress_start 信号后立即从 IDLE 进入 RESET 状态，进行四个中间变量在复位阶段的初始化；如果后续输入数据有效信号 data_valid 有效，则跳转到 UPDATE 进行更新阶段的计算；在计算过程中只要出现 data_valid 信号无效的情况，立即跳转到 UPDATE_WAIT 状态等待后续输入；如果在 UPDATE 状态下，处理计数器 proc_cnt 的值与待压缩数据流长度寄存器 length_reg 中的值差距小于 20 个字节，且中间变量更新状态 vi_cnt 值为 3，则跳转到 UPDATE_END 状态，并紧接着跳转到 DIGEST_WAIT，进入摘要阶段；为了保证摘要阶段计算的连续性，并减少缓存中间变量所需的寄存器，设置成当剩余的数据全部被写入到缓冲器后才开始进行摘要计算；摘要阶段根据 proc_cnt 与 length_reg 之间的差值来确定需

要跳转到的状态；完成最终的摘要计算 DIGEST_END1 和 DIGEST_END2 后，校验字计算完成，电路再次进入空闲 IDLE 状态，等待下一次计算请求。

3.2.7 流水线控制器

流水线控制器是整个 LZ4 压缩电路的主控逻辑，负责协调各个子电路的工作。流水线控制器本质上是一个有限状态机，其状态转移过程如图 3-14 所示。

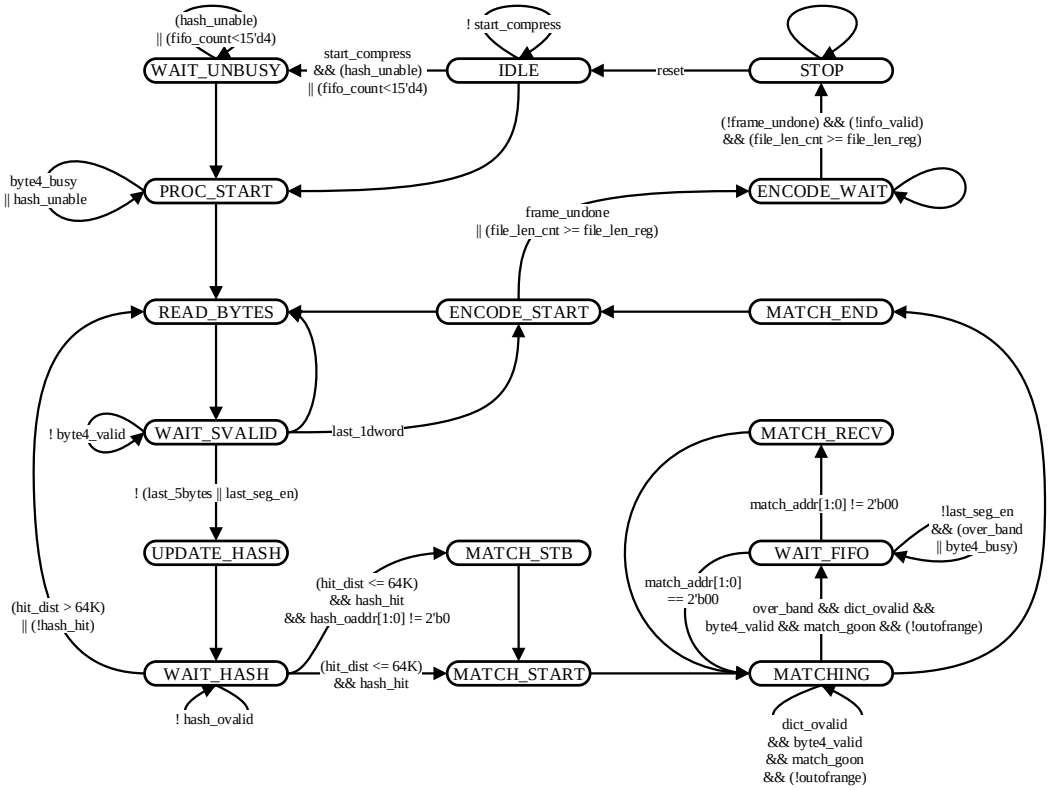


图 3-14 流水线控制器状态转移图

电路初始化为 IDLE 状态，当接收到外部 start_compress 信号后，离开 IDLE，此时如果匹配电路繁忙，即 Hash_unable 为高电平，又或者外部数据 FIFO 中的数据量少于 128Byte，即 fifo_count 值小于 4 时，则转移到 WAIT_UNBUSY 状态进行等待，直到 Hash_unable 为 0，fifo_count 不小于 4 时才转移到 PROC_START 开始首个 4 字符串的处理。

如果离开 IDLE 时 Hash 电路已空闲且外部 FIFO 中的数据量超过 128Byte，则直接转移到 PROC_START。在 PROC_START 状态下，如果分割电路或匹配电路处于繁忙状态，则维持该状态，直到分割电路和匹配电路均空闲时，转移到 READ_BYTES。

从分割电路读取数据时，从输入读请求信号到输出有效数据之间存在至少一个时钟周期的延迟，所以无条件转入 WAIT_SVALID 等待有效数据。一旦 byte4_valid 信号为高电平，代表此时来自分割电路的数据有效，判断当前剩余字符串长度是否少于 5 字节，分别转移到不同的状态。即 last_5bytes 或 last_seg_en 信

号为高电平时，如果剩余字节数为 0，last_1dword 为高电平，则转移到 ENCODE_START 开始最后的 5 字节编码，否则转移到 READ_BYTES 读出剩余所有字节；last_5bytes 和 last_seg_en 都为低电平时，剩余字符串多于 5 字节，转移到 UPDATE_HASH 开始更新 Hash 表。

读取和更新 Hash 表需要至少 1 个时钟周期的延迟，所以进入 UPDATE_HASH 状态后立即转移到 WAIT_HASH。一旦 Hash_ovalid 信号为高电平，则代表 Hash 表输出了可能存在的匹配数据和绝对首地址，此时流水线控制器将 Hash 表输出的 4 字符串与 READ_BYTES 状态下读出的 4 字符串进行对比，如果两字符串一致，则 Hash_hit 为高电平，代表找到了一次真匹配；然后计算两字符串绝对首地址之差 hit_dist，如果 hit_dist 不大于 65535（64K），则为有效匹配，根据输出 Hash_oaddr 地址是否 4 字节对齐分别转移到 MATCH_START 或 MATCH_STB 状态。

MATCH_STB 是为了补偿非 4 字节对齐时读取数据附加的一个时钟周期延迟，所以该状态会在下一个时钟周期无条件转移到 MATCH_START。确定了首个 4 字符串的匹配距离后，由 MATCH_START 转移到 MATCHING，进行扩展匹配。只有当字典缓冲器输出有效信号 dict_ovalid、分割电路输出有效信号 byte_valid、匹配继续信号 match_goon 均为有效高电平，且字典缓冲器地址溢出信号 outofrange、over_band 为无效低电平时，才能继续维持 MATCHING 状态，并进行扩展匹配。

如果除 over_band 信号外，其余信号均满足上述要求，则转移到 WAIT_FIFO 等待字典缓冲器电路，否则转移到 MATCH_END 终止扩展匹配。在 WAIT_FIFO 状态下，只要 last_seg_en 为低电平且 over_band、byte4_busy 信号中的任意一个为高电平，则维持 WAIT_FIFO 状态，否则根据字典中的匹配串绝对首地址 match_addr 是否 4 字节对齐，分别转移到 MATCHING 或 MATCH_RECV 状态继续扩展匹配。

扩展匹配完成后进入 ENCODE_START 状态进行压缩帧编码，如果帧编码未完成，即 frame_undone 为高电平，则直接转移到 ENCODE_WAIT，否则根据已处理字节数 file_len_cnt 和总字节数 file_len_reg 的相对大小转移到 READ_BYTES 进行末尾剩余字节读取，或转移到 ENCODE_WAIT 等待当前编码完成。当 frame_undone、info_valid 信号均为无效的低电平，且已处理字节数不小于总字节数时，代表一次完整压缩过程已结束，跳转到 STOP 状态。

在 STOP 状态下，只有异步复位信号 reset 可以将流水线控制器重新初始化，并开始下一次压缩过程。

3.3 FPGA 逻辑设计

采用自底向上的设计思想，使用 Verilog HDL 在 FPGA 上实现 3.2 节所述的 LZ4 压缩电路中的各个功能模块电路，然后连接组合成为 LZ4 压缩电路，经过逻辑综合后，总体电路如图 3-15 所示。其中对压缩电路性能影响较大的字典缓冲器和匹配电路的 FPGA 逻辑综合图分别如图 3-16 和图 3-17 所示。

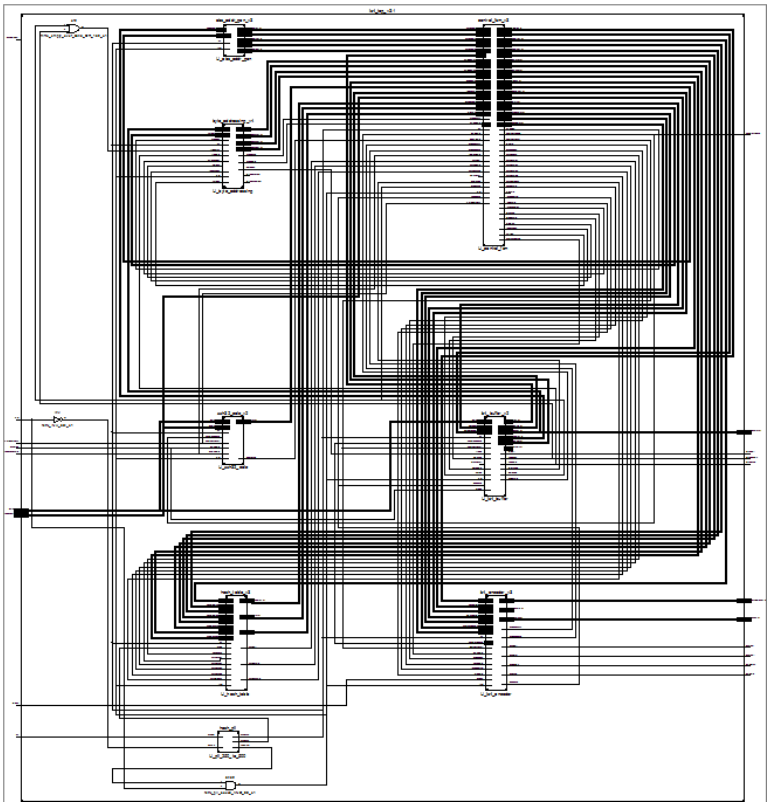


图 3-15 LZ4 压缩电路的 FPGA 逻辑综合图

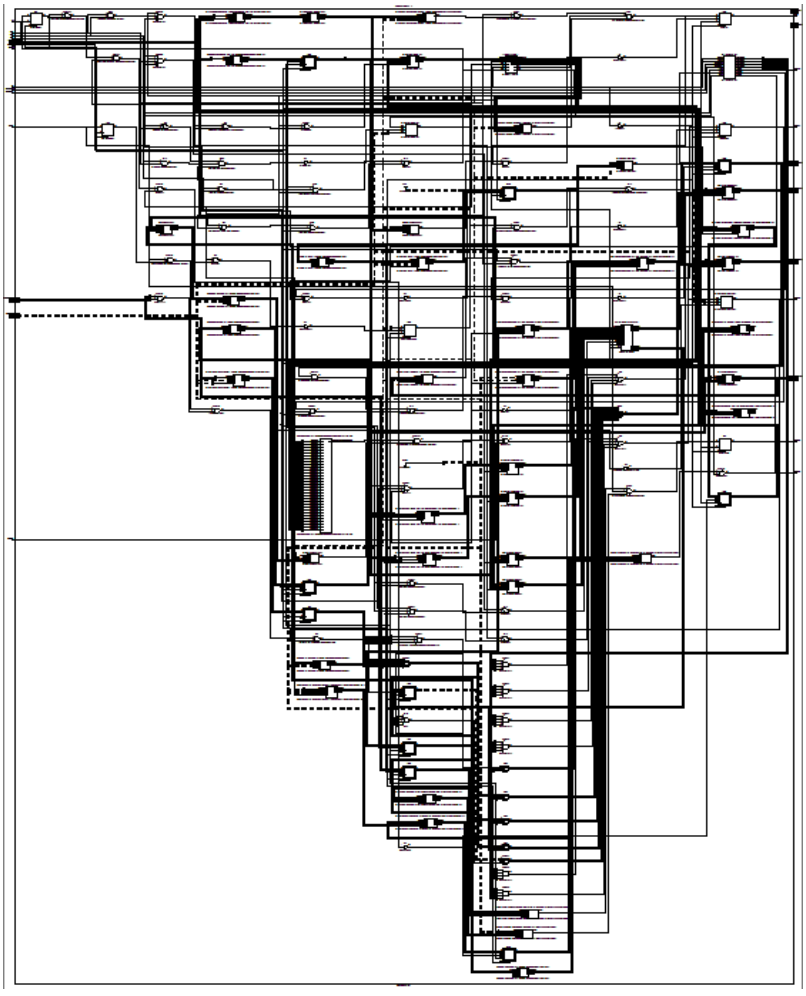


图 3-16 字典缓冲器的 FPGA 逻辑综合图

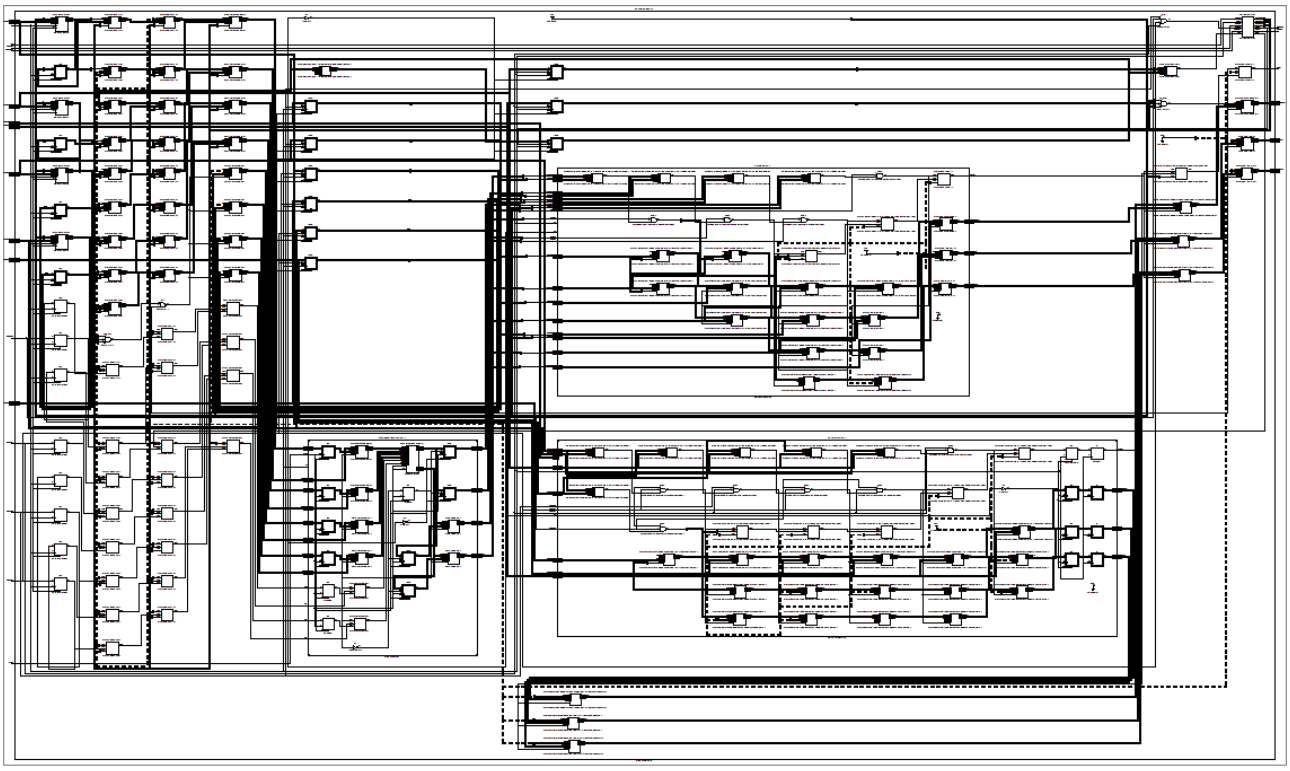


图 3-17 匹配电路的 FPGA 逻辑综合图

为了对 LZ4 压缩电路电路进行调试，针对 3.2 节所述的电路设计了对应的仿真文件，用于将待压缩的文件读入，组成数据流输入给电路进行仿真，仿真结束后，将结果读出，生成已压缩的文件。对已压缩文件进行解压缩，并与原始文件进行对比，找出其中不一致的部分，分析错误原因并回溯到电路的硬件代码中进行修改，直到解压缩后的文件与原始文件完全一致。电路仿真过程涉及大量的波形分析，本文不再赘述，这里给出仿真启动时，向 LZ4 压缩电路写入文件的波形，如图 3-18 所示，向电路发送 start_compress 信号，在紧接着的 3 个时钟周期内将待压缩数据长度 file_length 写入电路，并启动外部数据 idata 的输入，LZ4 压缩电路接收待压缩数据进行压缩处理。在压缩结束后，如图 3-19 所示，在接收到 compress_done 信号的上升沿之后，在一个时钟周期内将已压缩的数据从 LZ4 电路中取出，写入文件中，并记录当前压缩后的文件尺寸 compressed_len。

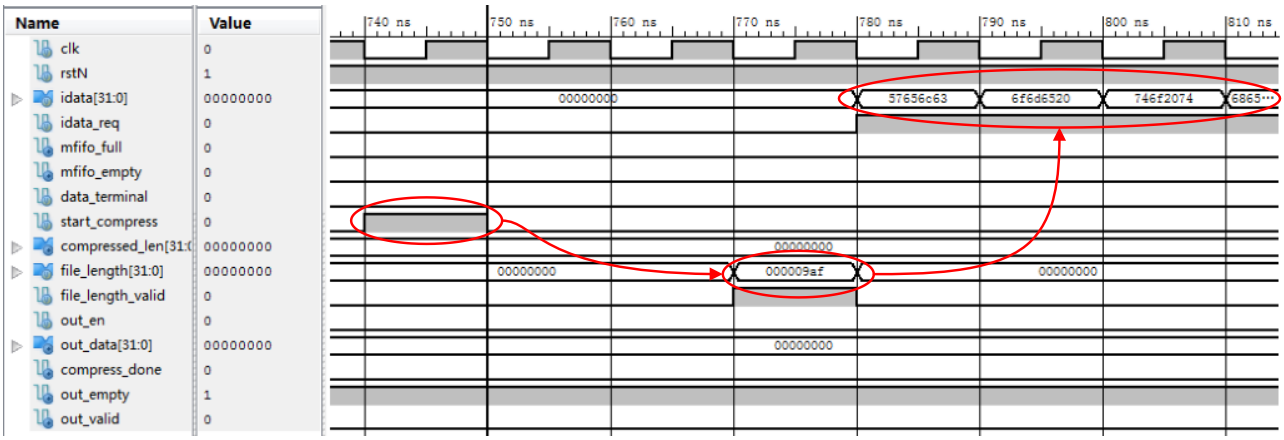


图 3-18 LZ4 电路仿真启动过程波形图

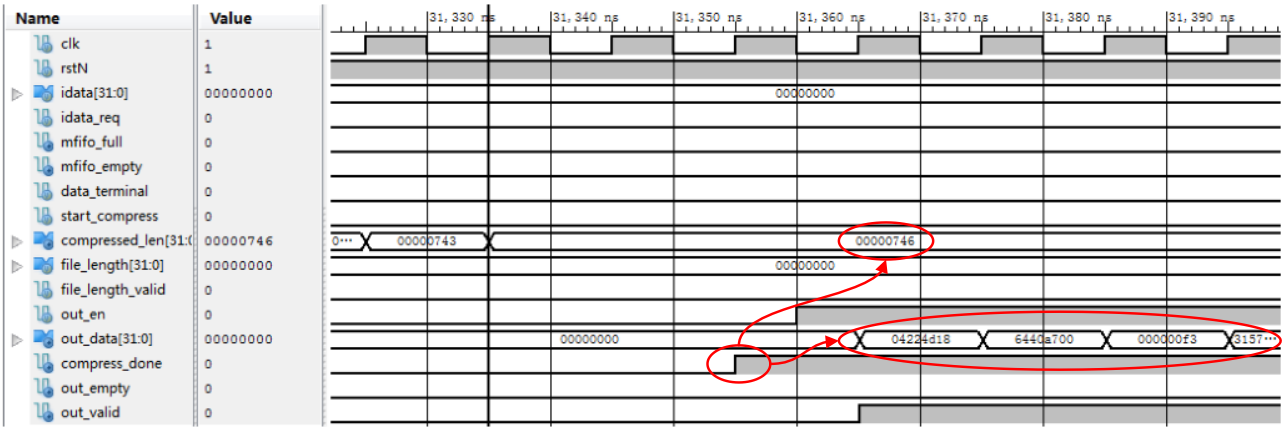


图 3-19 LZ4 电路仿真结束过程波形图

3.4 本章小结

本章提出了一种优化 LZ4 算法压缩率的方法，并设计了对应的压缩电路，其中包括可分时处理四种请求的字典缓冲器电路，旨在提升处理效率。匹配电路采用四路并行 Hash 值计算和写入机制，提升 Hash 表的更新频次，提升后续查找匹配的命中率。设计的字符串分割电路用于实现字符串的按字节移位。并行编码器和校验电路用于生成目标编码。流水线控制器用于控制 LZ4 压缩过程的并行化，以提升压缩速率。最后，使用 Verilog HDL 在 FPGA 上实现了 LZ4 压缩电路。

第四章 半静态 Huffman 编码电路设计

第四章针对原始 LZ4 压缩算法压缩率较差的问题提出了一种解决方案，结合静态 Huffman 编码速率高和动态 Huffman 编码压缩率性能高的特点，提出一种可用于 LZ4 输出数据格式的半静态 Huffman 编码压缩方法，并且设计了相应的硬件压缩电路。

4.1 半静态 Huffman 编码方法

如图 2-2 和图 2-3 所示， LZ4 压缩算法的输出是一种多帧格式的数据流，并且每一帧信息的组合方式非常紧凑，使其在输出格式上的冗余较 LZ77 等压缩算法要小很多。然而，LZ4 在搜索匹配的过程中，只去寻找与当前字符串最近的那个匹配串，相对于使用 Hash 链表来搜索字典中最长匹配的 LZ77 算法而言，LZ4 在匹配过程中的去冗余程度要差一些。这意味着 LZ4 压缩输出数据可以被二次压缩，以获得更好的压缩率。

半静态 Huffman 编码是由 Danny H, Ety K, Dmitry S 等人[错误!未找到引用源。]提出的一种适用于高压缩速率场合的熵编码思想。其在处理速率和压缩率方面作了折中，用长度固定的部分统计替换动态统计，以节约处理时间，提升了编码速率，并且，由于其静态编码阶段使用的是与当前待压缩源的字符分布相似的变长编码表进行编码，所以压缩率方面较为接近动态 Huffman 编码的结果，而压缩速率较为接近静态 Huffman 编码[错误!未找到引用源。]。统计长度是预先设置的，待统计数据越长，耗时越长，统计结果越接近信源分布，压缩率也越好。

本文所述的半静态 Huffman 编码方法及其电路实现既可用于 LZ4 压缩输出数据的二次压缩，也可用于其它以字节为单位的数据压缩。其编码流程如图 4-1 所示。以统计长度为 8KB 的数据进行半静态 Huffman 编码（以下简称为“部分统计”）为例，完整的编码过程可以分为如下三个阶段：

(1) 统计阶段：首先获取待压缩数据尺寸，然后对待压缩数据或其中前 8KB 数据进行统计，将每个位宽为 8bit 的字符出现的次数记录下来，最多有 256 个记录，如果小于 8KB，则直接按照动态 Huffman 编码的方法来进行编码，如果大于 8KB，则按照半静态 Huffman 编码的方法编码；

(2) 建树阶段：首先初始化生成 256 个不同编号从 0 到 255 的叶节点，将统计阶段获得的各字符出现次数分别作为对应编号叶节点的权重，如果某个编号的叶节点没有对应的出现次数记录，则将其权重置为 0，不作为二叉树的叶节点；然后对所有权重不为 0 的叶节点按各自的权重进行堆排序，获得权重递增序列；从序列最左边取出 2 个权重最小的节点，将其合并成为一个新节点，权重为之前两个节点权重之和，并将新生成的节点按其权重大小插入到递增序列中，并重新从序列中取出两个权重最小的节点，以此类推，直到整个序列中只剩下一个节点时，建树阶段完成。

(3) 编码阶段：首先根据上一阶段建立的二叉树，依次从各个叶节点向根节点方向进行扫描，计算各叶节点到根节点的路径上一共多少个中间节点和根节点，分别作为每个叶节点对应字符的初始码长，如果某些字符的叶节点不属于二叉树，则将其初始码长置为 0；然后对各字符的初始码长进行优化，使各码长被控制在 1bit 到 15bit 之间；最后使用优化后的码长生成变长编码表，对待压缩数据的前 8KB 字符和之后的所有字符编码进行编码，实现压缩功能。

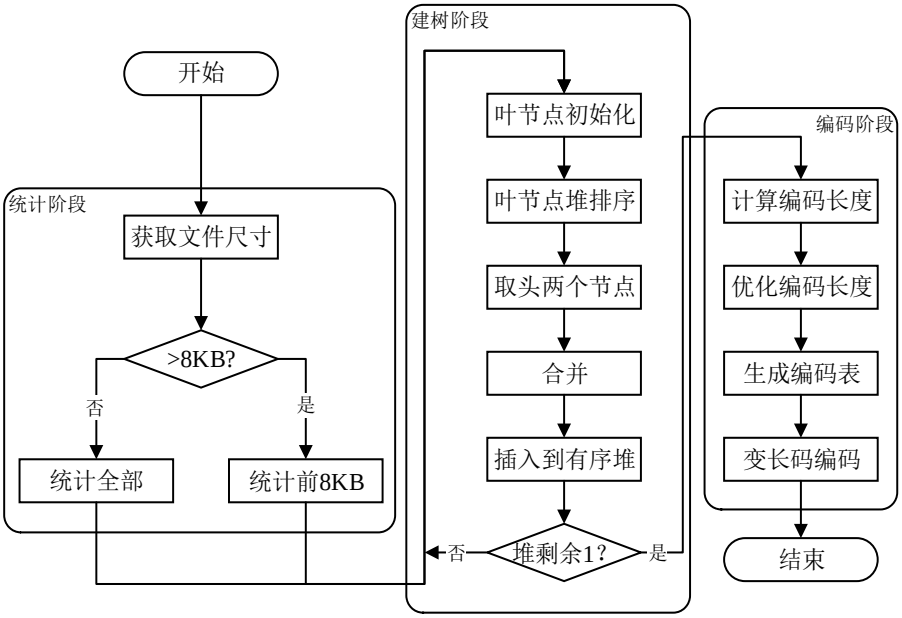


图 4-1 半静态 Huffman 编码流程图

4.2 半静态 Huffman 编码电路设计

根据 4.1 节所述的半静态 Huffman 编码流程，以及硬件电路并行化特点，设计半静态 Huffman 编码电路的架构和各模块电路。

4.2.1 总体架构

如图 4-2 所示，半静态 Huffman 编码电路由一个六级流水线、一个有序序列存储器、一个码长查找表存储器以及一个原始数据存储器构成。

其中，六级流水线由统计电路、排序电路、建树电路、码长优化电路、码表生成电路以及编码电路构成。统计电路接收来自外部的输入数据，并且以字节为单位，统计不同的各个字符出现的次数，将统计信息通过 RAM 接口交给下一级排序电路，同时将输入数据放入原始数据存储器。排序电路按照统计信息中各个字符出现的次数，将各字符的编号连同统计次数构建成节点帧，并排列成为有序序列，最终将序列通过 FIFO 接口存入有序序列存储器。建树电路通过 RAM 接口依次读取排序电路中的有序序列，并按照二叉树的建树规则，生成叶节点，中间节点和根节点，最终生成各节点之间的链接关系。码长优化电路通过 RAM 接口获取二叉树信息，统计各叶节点对应的编码长度，然后使用码长优化算法对最大码长进行限制，

最终生成与字符一一对应的码长查找表，通过两个 RAM 接口分别将查找表传递给码长查找表存储器和下一级电路。码表生成电路使用经过优化的码长查找表生成与各个字符一一对应的变长编码表。编码电路使用变长编码表对原始数据存储器中的字符进行替换和拼接，并加入解码所需的信息，最终通过 FIFO 接口向外部送出宽度固定的已编码输出数据。流水线中各级电路将在后面几节详细介绍。

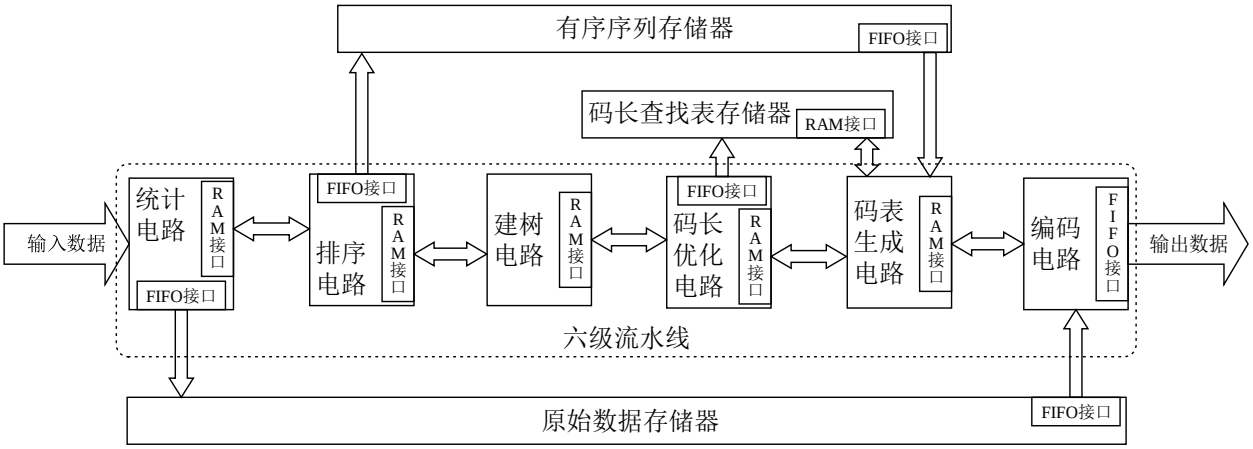


图 4-2 半静态 Huffman 编码电路框图

有序序列存储器和码长查找表存储器用于保存半静态 Huffman 编码过程中的关键信息，这些信息被用于生成变长编码表，并在编码时被放置在输出数据的头部，便于 Huffman 解码器恢复出变长编码表，从而对后面的数据进行 Huffman 解码。为了防止流水线处理过程中，原始数据被后续的数据覆盖，使用原始数据存储器对输入数据流进行缓冲和延迟，使原始数据在一个完整的六级流水线处理周期中得以保留。

4.2.2 统计电路

统计电路用于记录输入的 4Bytes 数据流中，各 8bit 字符出现的次数。如图 4-3 所示，统计电路包含 4 个累加计数的计数器组（Counter Group，CG）和相应的控制逻辑。字符串分割器用于将输入的 4Bytes 数据流分解成 4 个 Byte 字符流，对每个字符出现的次数进行相互独立的累加计数，因为 8bit 二进制数组成的字符一共有 256 种可能，所以每个计数器组包含 256 个相互独立的计数器。整个统计电路一共包含 4 个并行的计数器组 CG_1、CG_2、CG_3、CG_4，每个时钟周期能够并行处理 4 个字符，可对输入数据进行实时同步的统计。统计完成后，任意一个字符出现的总次数是 4 个并行的 CG 中对应计数器值之和，其输出将被四输入加法器叠加。并且，统计电路中包含自动清零逻辑，用于在后级电路完全将 4 个 CG 中的数据读出后，重新初始化统计逻辑，并将 4 个 CG 中的所有计数器值清零，等待下一次统计操作。模块内部针对 CG 的读取和清零等控制信号来自于 RAM 接口控制器，受控于后级电路。

考虑到使用大量触发器实现计数器组会导致大量消耗逻辑布线空间，造成数字电路的时序性能受限，故使用 18bit 位宽，地址空间 0~255 的双端口 RAM 来实现单个计数器组，最大可统计 256K 个字符。如图 4-4 所示，在统计阶段，A 端口以输入 8bit 数据 data 为地址，读取累加前的计数器值，B 端口把延迟一个

时钟周期的 data_delay 作为地址，延迟后的数据有效信号 data_valid 作为写使能信号，写入累加后的计数器值；在排序阶段，受选通信号 sort_en 控制，A、B 端口均用于向排序电路输出计数器值，使用外部输入 rd_addra 和 rd_addrb 地址同时读取 RAM 内的计数值，以减少 CG 转移数据所需的时间；在清除阶段，受选通信号 clean_en 控制，A、B 端口均用于写入初始化计数值 0，写入地址来自递增计值 clean_cnt_1 和 clean_cnt_2，分别从 0 和 128 开始计数，写使能信号为 clean_valid，每个时钟周期清除 CG 中的两个计数器。

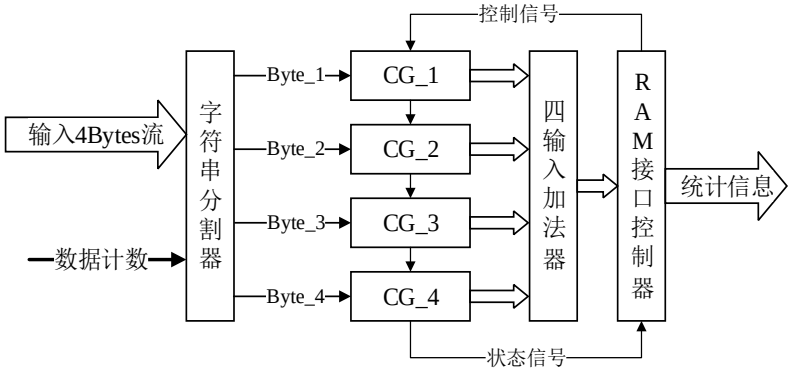


图 4-3 统计电路框图

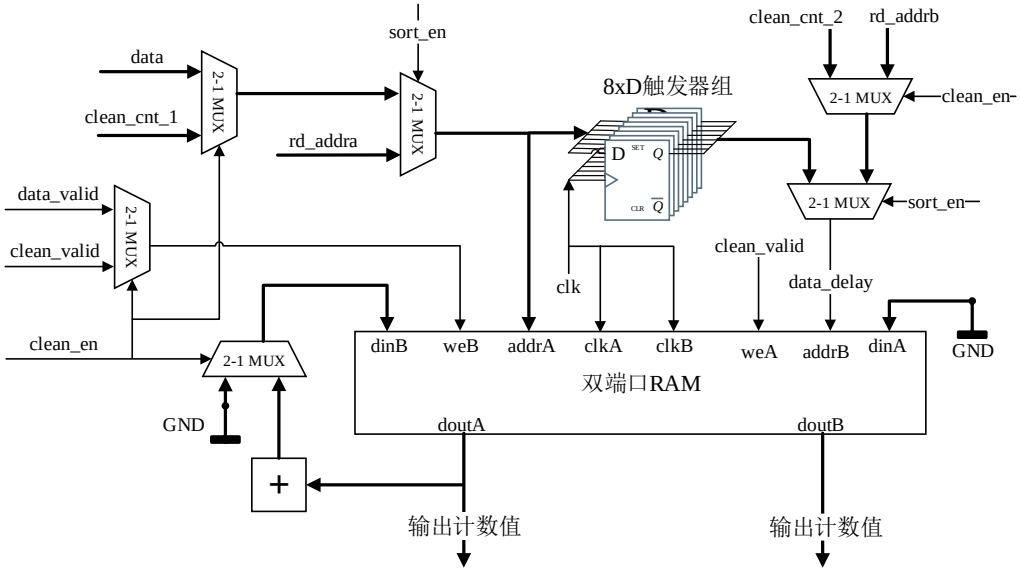


图 4-4 双端口 RAM 实现计数器组原理图

4.2.3 排序电路

排序电路按各字符出现的次数进行排序，使所有字符按照权重大小组成一个严格有序的序列。如图 4-5 所示，排序电路中包含两个位宽 36bit，深度为 128 的 RAM，分别被称为奇 RAM 和偶 RAM。奇、偶 RAM 用于存储在排序过程中按权重升序排列的叶节点和相应的叶节点编号，存储的节点帧格式为编号在高 9bit，权重在低 27bit。堆排序逻辑将上述的两个 RAM 作为缓存，从统计电路读取统计信息，组合成节点帧后，对其按权重值大小进行排序。排序结束后，奇、偶 RAM 中各存储一半的有序序列，通过 FIFO 接

口依次将有序序列存入外部的有序序列存储器中，并提供随机访问有序序列的 RAM 接口供后级电路使用。

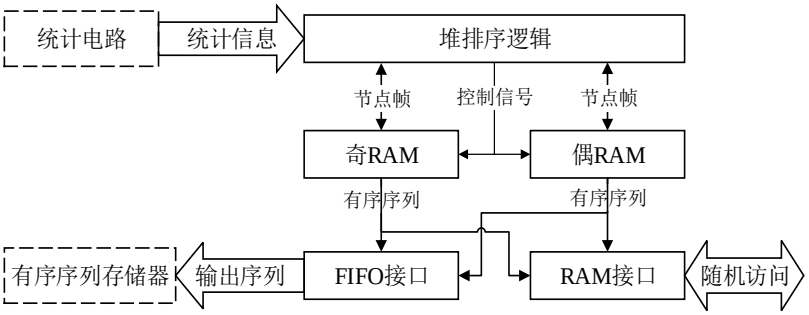


图 4-5 排序电路框图

堆排序的逻辑状态转移过程如图 4-6 所示，无论是在空闲状态 IDLE 还是上一次排序完成状态 SORT_DONE，只要排序电路接收到起始信号 heap_sort_start，则进入 INIT 状态初始化奇、偶 RAM，将来自上一级统计电路的统计结果按照每个时钟周期 2 个数据的时序，组成两个节点帧，分别写入两个 RAM 对应的地址单元中。并且，初始化时保证每个时钟周期取出的较大的数据所在的节点帧被存入了奇 RAM，而较小的数据被存入了偶 RAM，如果两个数据值相同，则将节点号较大的帧存入奇 RAM 中。初始化过程共消耗 128 个时钟周期。每次堆搜索过程由 SEG_SORT、SEG_WAIT、SEG_DONE 三个状态来完成，每对小根堆进行一次扫描排序需要消耗 128 个时钟周期，并保证相同地址下的奇 RAM 中的节点权重大于偶 RAM 中的节点权重。如果小根堆的无序区计数 heap_size 已等于 1，代表排序完成，此时状态转移到 HEAP_OUT，将排序完成的序列存入排序电路外部的有序序列存储器中。待有序序列转移到建树电路后，排序电路进入 SORT_DONE 状态以表明排序已完成，等待下一次起始信号的到来。

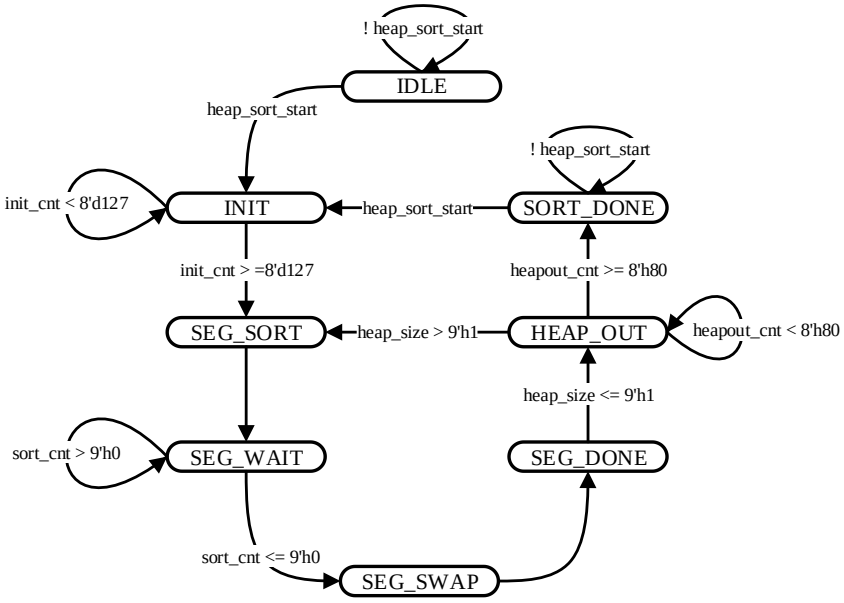


图 4-6 堆排序逻辑状态转移图

实现排序电路的关键是实现堆排序逻辑，图 4-7 描述的是堆排序逻辑中使用的排序单元结构，使用 3 个两输入两输出比较器和 3 个缓冲器实现三输入三输出的小型排序网络。其中输入 in_1、in_2、in_3 是 3 个待排序的 36bit 节点帧，输出 max 是低 27bit 权重最大的节点帧、min 是权重最小的节点帧、mid 是剩余

的节点帧。如果在排序过程中有 2 个或 2 个以上的节点帧权重相等，则将低 27bit 权重相等的节点帧按照高 9bit 的节点号重新排序。

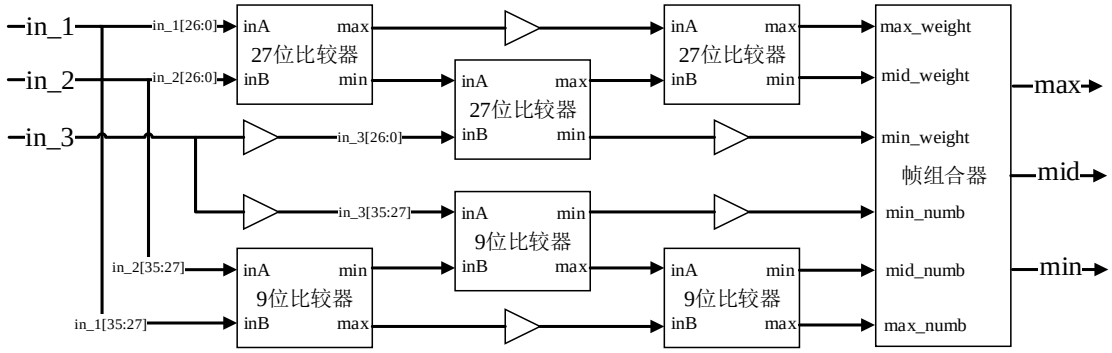


图 4-7 三输入三输出节点帧排序网络

4.2.4 建树电路

建树电路按照各字符的权重，生成用于编码的二叉树链接信息。如图 4-8，建树电路中包含两个位宽 10bit，深度 256 的 EVEN_RAM 和 ODD_RAM，分别用于存储偶数编号和奇数编号的节点指针。基于 Huffman 编码所使用的二叉树的性质，在建树过程中需要存储的节点总数不超过叶节点数的 2 倍，使用 2 个深度 256 的 RAM 就足以存储所有节点指针。建树逻辑用于从前一级排序电路中获取有序的节点帧，并在节点合并后向排序电路写入新生成的中间节点帧。由于建树过程中需要保证排序电路中节点帧序列的有序性，所以，在建树逻辑中还包含对加入中间节点后的序列进行排序的功能。建树逻辑最终能够建立从各叶节点指向中间节点直到根节点的指针，指针的存储形式是各节点对应的父亲节点的编号。建树电路向后级电路提供随机访问 EVEN_RAM 和 ODD_RAM 的接口。

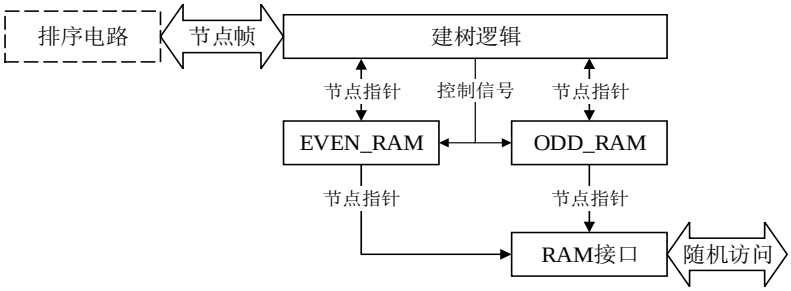


图 4-8 建树电路框图

建树逻辑的状态转移过程如图 4-9 所示，无论电路在空闲状态 IDLE 还是上一次建树完成状态 TREE_DONE，只要接收到起始信号 heap_sort_start，则进入读取节点状态 NODE_READ，从上一级排序电路的有序序列头读取两个权重最小的叶节点。

读取到节点后，进入 NODE_MERGE 对两个叶节点进行合并，读出的叶节点中可能有权重为 0 的情况，此时，该叶节点所代表的字符在统计过程中没有出现过，建树时跳过即可。如果读出的两个叶节点的权重全为 0，则转移到 NODE_SKIP 状态，跳到下一个地址继续读取；如果只有一个叶节点权重为 0，则转

移到 READ_NODE 状态再补充读一个叶节点；如果所有叶节点权重均不为 0，则分配新的节点号，并转移到 NODE_SEEK 来查找合并后的中间节点在有序序列中的插入位置，根据搜索完成信号 seek_done 来判断是否完成搜索，并转移到 NODE_INSERT 将合并生成的中间节点插入到有序序列中，并将合并后分配的节点号写入用于合并的两个原始节点所对应的 RAM 单元中。

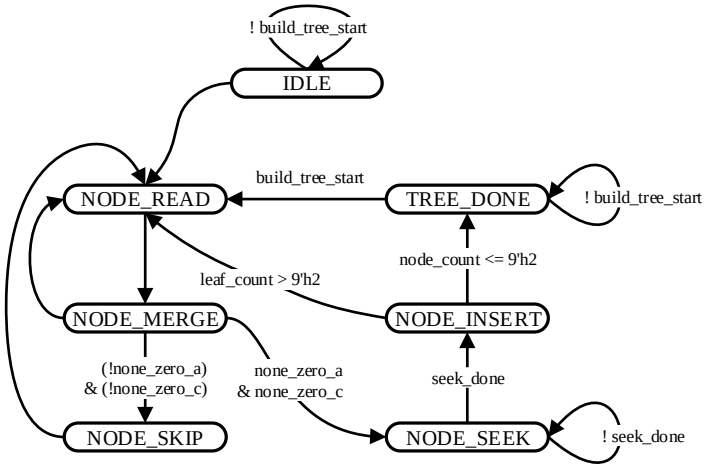


图 4-9 建树逻辑状态转移图

最后，判断节点计数器 node_count，如果值为 2，则代表有序序列中只剩下 2 个节点了，建树过程已经到达根节点，转移到建树完成状态 TREE_DONE，将最后两个指针写入相应的 RAM 单元中，完成整个建树过程。

4.2.5 码长优化电路

码长优化电路用于限制二叉树的最大深度，从而限制变长编码的最大码长。如图 4-10 所示，码长优化电路包含三个计数器组 BL_COUNT_1、BL_COUNT_2 和 OPT_COUNT，以及对应的扫描优化逻辑和码长查找表生成器。扫描优化逻辑通过 RAM 接口读取前级建树电路中 EVEN_RAM 和 ODD_RAM 中存储的节点指针信息，同时扫描两条从叶节点到根节点的路径，路径中每经过一个中间节点或根节点，其路径长度自增 1，扫描到根节点即获得当前叶节点对应字符的初始码长，然后将该码长对应的计数器值自增 1，并将初始码长写入对应字符的外部码长查找表存储器单元中。由于最大码长被限制为 15，不同的字符类型有 256 种，所以每个计数器组包含 16 个 9bit 计数器，用于存储码长从 0 到 15 的字符个数，如果某字符的码长超过 15，则当作码长为 0 对待。在码长扫描过程中，计数器组 BL_COUNT_1 和 BL_COUNT_2 用于存储码长 1 到 15 的正常节点个数，以及码长为 0 或超过 15 的溢出节点计数。

对所有字符的码长扫描完成后，扫描优化逻辑根据图 4-11 所示的流程图对初始码长进行优化。其中 bl_count[i]代表 BL_COUNT_1 和 BL_COUNT_2 中第 i 个计数器的值之和，bl_count[0]是溢出节点的数量，从树的第 14 层开始寻找可插入溢出节点的位置，如果 14 层没有叶节点，则逐层递减直到找到叶节点。bl_count[i]自减 1 本质上是将叶节点从二叉树中分离，bl_count[i+1]自增 2 本质上是将前一步骤被分离的叶

节点下移一层，然后与需要添加进二叉树的溢出节点组成兄弟，新建父亲节点并插入二叉树中。整个优化过程持续到 bl_count[0]计数为 0，即所有溢出节点均已插入二叉树中，优化完成。

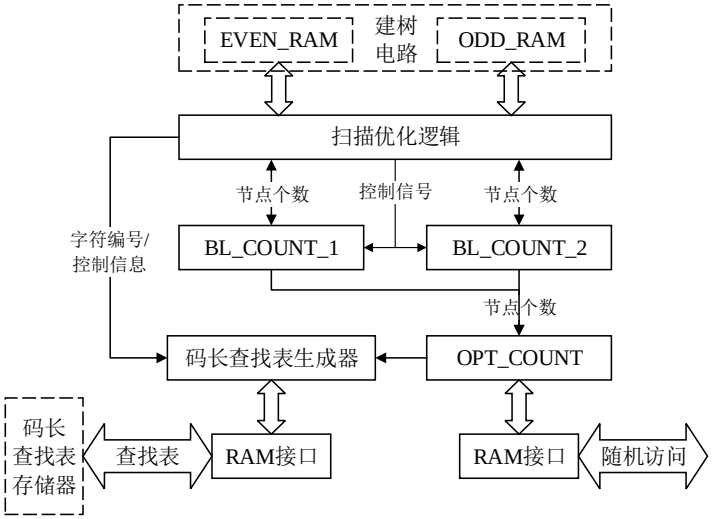


图 4-10 码长优化电路框图

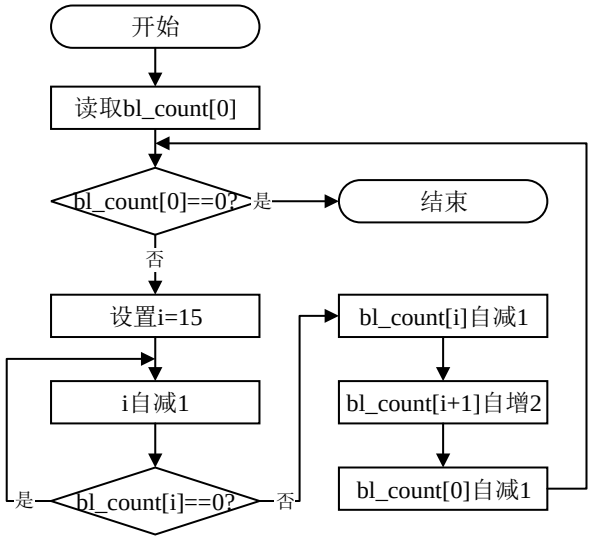


图 4-11 码长优化流程图

优化过程中，OPT_COUNT 用于存储码长 1 到 15 的节点个数和溢出节点个数。码长查找表生成器根据 OPT_COUNT 里的节点个数以及来自扫描优化逻辑的字符编号和控制信息，从 RAM 接口更新码长查找表存储器中对应的正常节点和溢出节点的码长。优化完成后，OPT_COUNT 中码长为 0 的溢出节点个数应为 0，外部码长查找表存储器内的所有值都应该在 1 到 15 的范围内。额外的，码长优化电路向后级电路提供可随机访问 OPT_COUNT 计数器组的 RAM 接口。

4.2.6 码表生成电路

码表生成电路用经过优化的码长查找表、有序序列等信息生成与各字符一一对应的变长编码，构建字符与变长编码之间的查找表。如图 4-12 所示，码表生成电路包括编码逻辑、包含 16 个 16bit 计数器的 NEXT_CODE 计数器组，以及一对互为镜像的变长编码表 VLC_TABLE_1 和 VLC_TABLE_2。编码逻辑从

码长优化电路中的 OPT_COUNT 计数器组顺序读出码长从 1 到 15 的节点个数，并按照式（4.1）计算 15 种不同码长的编码起始值，从而保证变长编码的独特可译性。

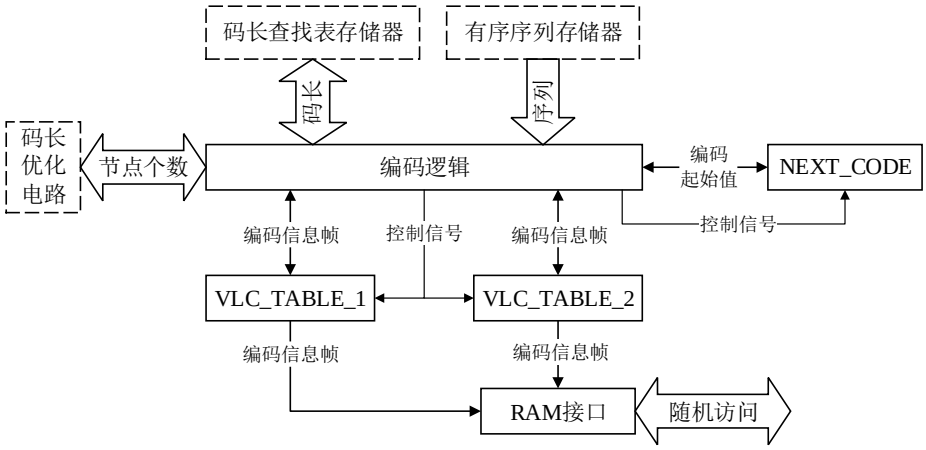


图 4-12 码表生成电路框图

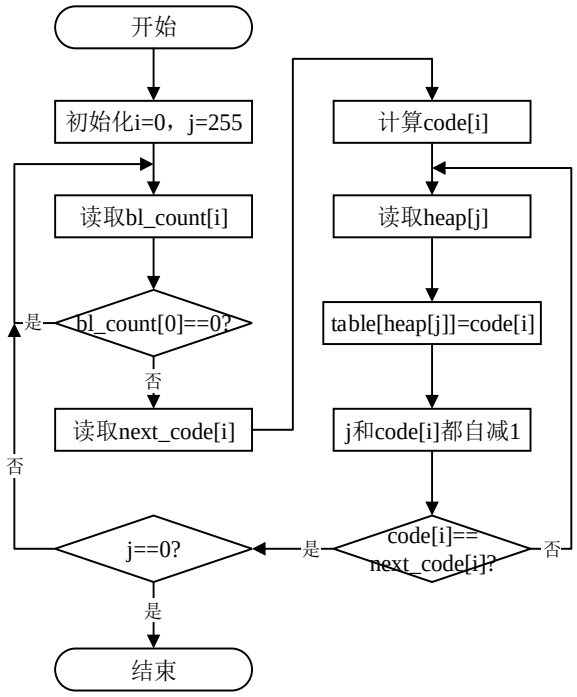


图 4-13 编码表生成流程图

式（4.1）中，$next_code[k]$代表码长为 k 的编码起始值，$bl_count[i]$代表码长为 i 的字符个数，并且所有字符中最短的码长对应的起始值是 0。

$$\begin{cases} next\_code[k]=0 & k=1 \\ next\_code[k]=\sum_{i=1}^{k-1} bl\_count[i]\times 2^{i} & k>1 \end{cases} \tag{4.1}$$

在硬件电路中，任意数值与 2 幂次的乘法可以通过二进制移位实现。由于码表生成过程是按照码长顺序进行的，所以累加过程可以通过保存前一个累加值的方法来依次计算。硬件电路中计算起始值的方法如式（4.2）所示。

$$\begin{cases} next\_code[k]=0 & k=1 \\ next\_code[k]=(next\_code[k-1]+bl\_count[k-1])<<1 & k>1 \end{cases} \tag{4.2}$$

确定编码起始值后，编码逻辑读取外部有序序列存储器，从小到大依次取出 256 个字符。每取出一个字符，就同时从外部码长查找表存储器中取出对应字符的码长，并将该码长对应的编码起始值和已被处理的相同码长的字符个数叠加作为当前字符的码字。其流程如图 4-13 所示，其中 $code[i]$ 代表码长为 i 的变长编码码字，计算方法见式（4.3）。

$$code[i] = bl\_count[i] + next\_code[i] \tag{4.3}$$

最后，将 1 到 15 的码长作为高 4bit，对应的变长编码码字作为低 16bit，合并成 20bit 的编码信息帧，同时存入深度为 256，位宽为 20bit 的 VLC_TABLE_1 和 VLC_TABLE_2 两个编码表之中，并向后级电路提供可随机访问的 RAM 接口。

4.2.7 编码电路

编码电路以各字符作为查表索引，使用变长编码查找表中的信息替换原始数据中的各字符，并拼接组合成固定宽度的编码数据流。

如图 4-14 所示，编码电路中包含四组查表逻辑 LU_1、LU_2、LU_3、LU_4 和一个变长码拼接逻辑。查表逻辑以 8bit 待压缩字符为索引，通过 RAM 接口从上级码表生成电路中读取对应的变长编码信息帧，并将码长、溢出标志位和编码值存入查表逻辑内部的 20bit 位宽，深度为 16 的缓冲队列中。四路并行的 LU 使编码电路在每个时钟周期内可以同时处理 4 个以 Byte 为单位的字符。变长码拼接逻辑接收四路编码信息帧，将各个码长不同的编码进行分割和拼接，组合成为宽度 4Bytes 的输出数据流。

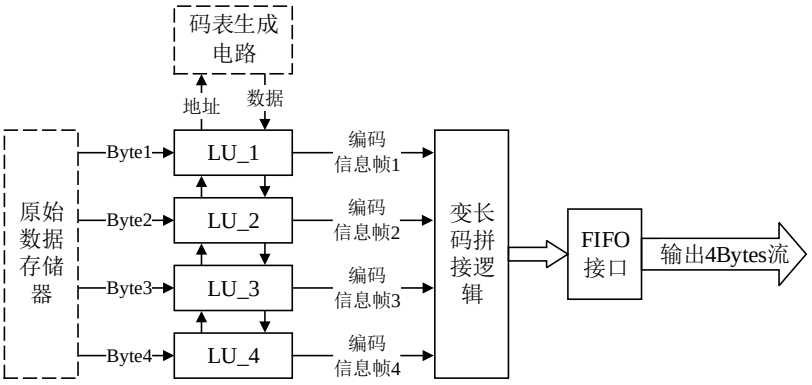


图 4-14 编码电路框图

变长码拼接逻辑的状态转移过程如图 4-15 所示。拼接逻辑在 IDLE 或 DONE 状态下接收到 vlc_glue_start 信号后，进入 INIT 状态并读取查表逻辑 1 中的首个变长码，如果查表逻辑 LU_1 输出了有效数据，则转移到初始化完成状态 INIT_DONE；如果此时 LU_2 中有数据，则读取 LU_2 并转移到变长码 1 和 2 的拼接状态 GLUE_12，否则，进入 LU_2 等待状态 WAIT_2，直到 LU_2 数据不为空，才转移到 GLUE_12；然后依次判断 LU_3、LU_4 是否为空，并转移到相应的等待 WAIT_3、WAIT_4、WAIT_1 或拼接状态 GLUE_23、GLUE_34、GLUE_41；在任何一个拼接状态中，一旦检测到所有编码已完成信号 all_done，则

直接转移到 GLUE_LAST 状态进行最后一个变长码的拼接，并转移到最后一次写入状态 WRITE；所有操作都完成后，转移到 DONE 状态，等待下一次拼接过程。

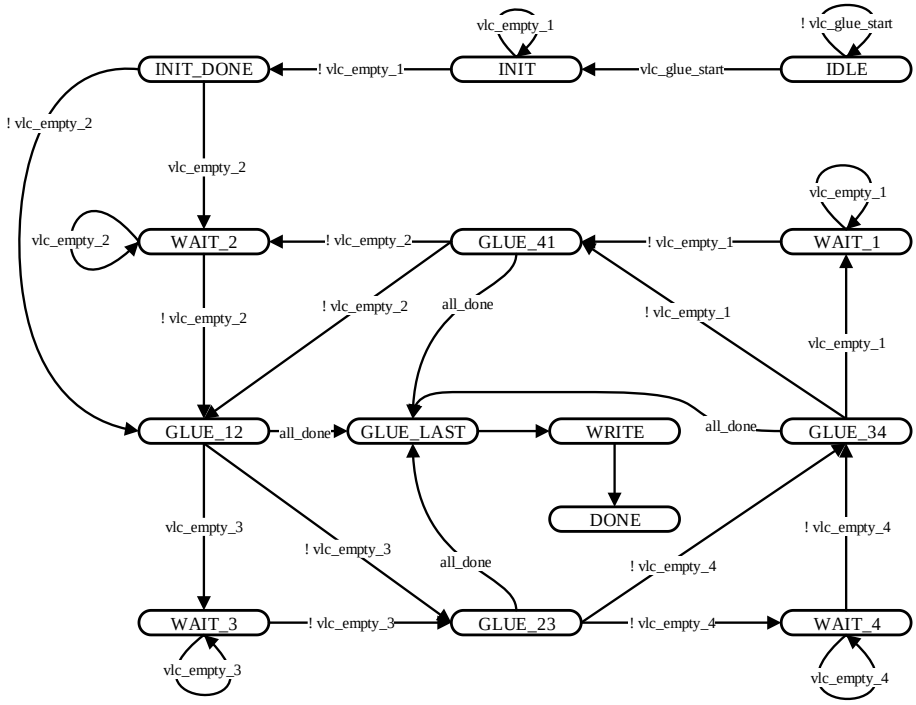


图 4-15 变长码拼接逻辑状态转移图

由于传统的变长码拼接方法需要很多移位操作，在电路实现的过程中会消耗大量的时钟周期，所以在本文提出的编码电路中使用的一种溢出控制写入电路来并行实现变长编码的拼接，每个时钟周期最多处理 2 个 20bit 编码信息帧，对其中的变长码字进行拼接操作。溢出控制写入电路如图 4-16 所示，其中溢出计数器接收到前一级累加器输出值大于或等于 32 时，产生溢出进位信号，作为编码输出 FIFO 的写使能信号 wr_en，与此同时，编码拼接器输出宽度固定为 32bit 的数据流，作为 FIFO 的写入数据。编码输出 FIFO 用于缓冲来自编码电路的输出数据，并起到降低半静态 Huffman 编码电路与外围电路之间耦合程度的作用。

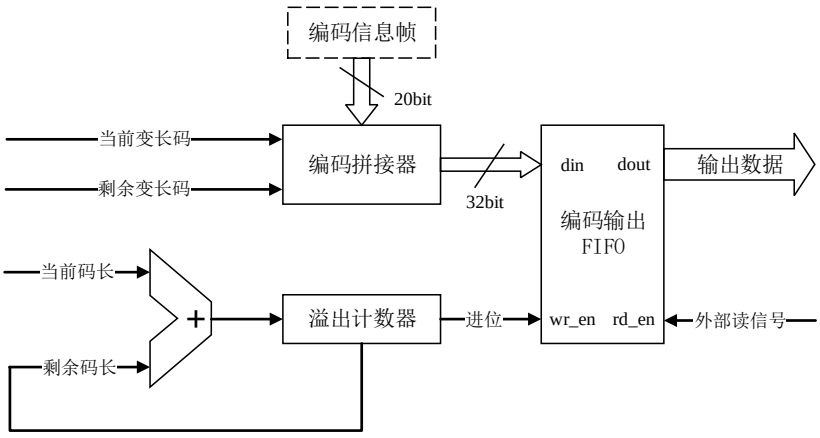


图 4-16 溢出控制写入电路

4.3 压缩电路级联

通过 LZ4 压缩电路和 Huffman 编码电路的级联，既保证了经过优化的 LZ4 无损压缩电路与 LZ4 压缩软件之间的兼容性，又在优化 LZ4 无损压缩算法电路的压缩率性能方面提供了可能性。级联后的电路根据外部选择不同的工作模式，分别切换到单级压缩或两级压缩的工作状态。

LZ4 压缩电路和半静态 Huffman 编码电路的输入和输出接口均采用了耦合度较低的 FIFO 方式实现，且输出数据宽度固定为 32bit，方便电路之间级联。按照两级压缩的思路，将 LZ4 压缩电路作为第一级，半静态 Huffman 编码电路作为第二级，构成如图 4-17 所示的两级压缩电路。其中，为了兼容原始 LZ4 无损压缩算法及其压缩软件的编码方式，加入了用于输出编码切换的两路选择器，当输出模式选择信号（mode_sel）为低电平时，两级压缩电路的输出是来自 LZ4 压缩电路的兼容编码流，此时仅 LZ4 压缩电路工作，半静态 Huffman 编码电路、级联逻辑和缓冲 FIFO 都不工作；相反的，当 mode_sel 高电平时，输出为来自半静态 Huffman 编码电路的优化编码流，此时 LZ4 压缩电路、级联逻辑、半静态 Huffman 编码电路以及缓冲 FIFO 全都正常工作。

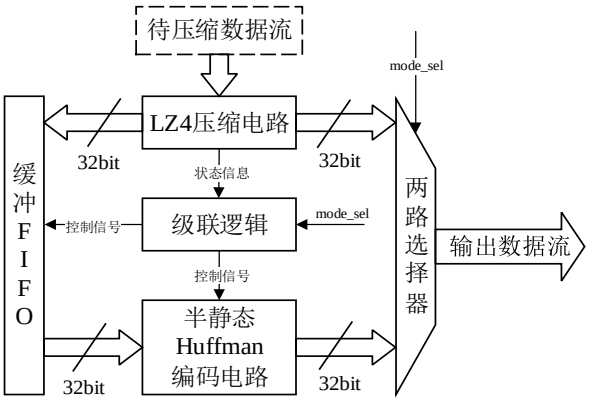


图 4-17 两级压缩电路

4.4 FPGA 逻辑设计与仿真

采用自底向上的设计思想，使用 Verilog HDL 在 FPGA 上实现 4.3 节所述的半静态 Huffman 编码电路的各个功能模块电路，然后连接组合成为 Huffman 编码电路，并设计级联逻辑，最终实现两级压缩电路。经过逻辑综合后，半静态 Huffman 电路的 FPGA 逻辑综合图如图 4-18 所示，两级压缩电路的逻辑综合图如图 4-19 所示。

为了对两级压缩电路电路进行调试，针对 4.3 节所述的电路设计了仿真文件，用于将待压缩的文件读入，组成数据流输入给电路进行仿真，仿真结束后，将结果读出，生成已压缩的文件。对已压缩文件进行解压缩，并与原始文件进行对比，找出其中不一致的部分，分析错误原因并回溯到电路的硬件代码中进行修改，直到解压缩后的文件与原始文件完全一致，仿真过程结束。由于电路仿真过程涉及大量的波形分析，

本文不再赘述，这里给出仿真过程的启动波形和结束波形分别如图 4-20 和图 4-21 所示。

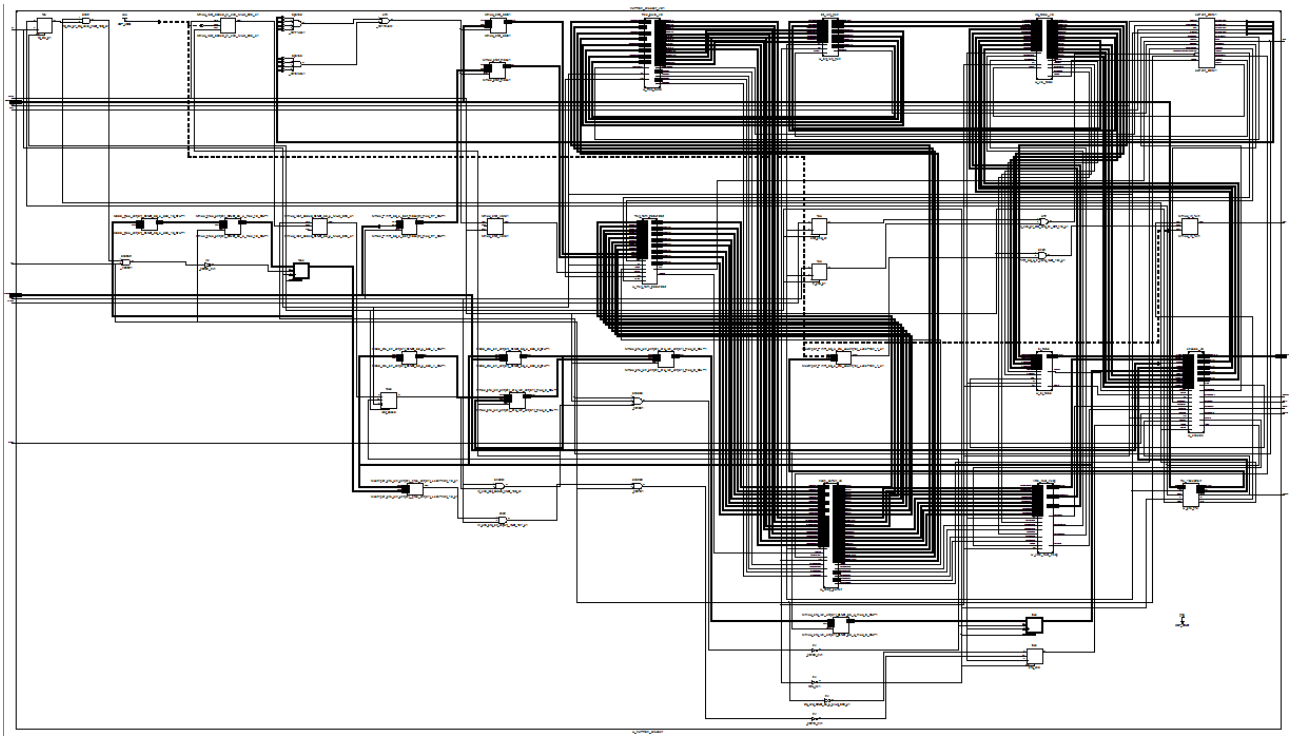


图 4-18 半静态 Huffman 电路的 FPGA 逻辑综合图

图 4-19 两级压缩电路的 FPGA 逻辑综合图

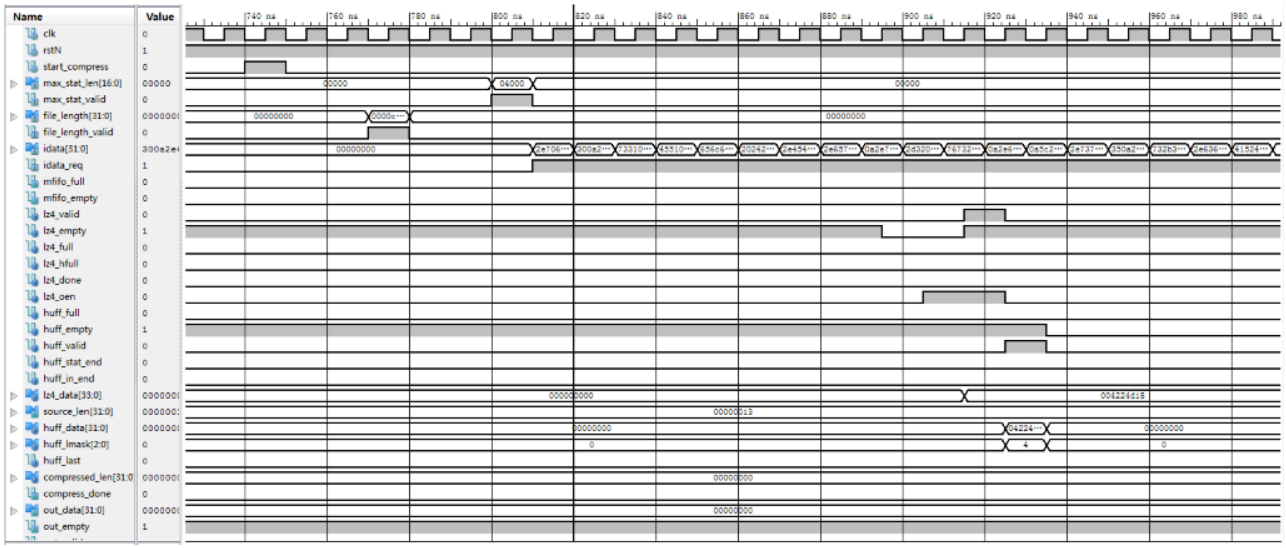


图 4-20 两级压缩电路仿真启动过程波形图

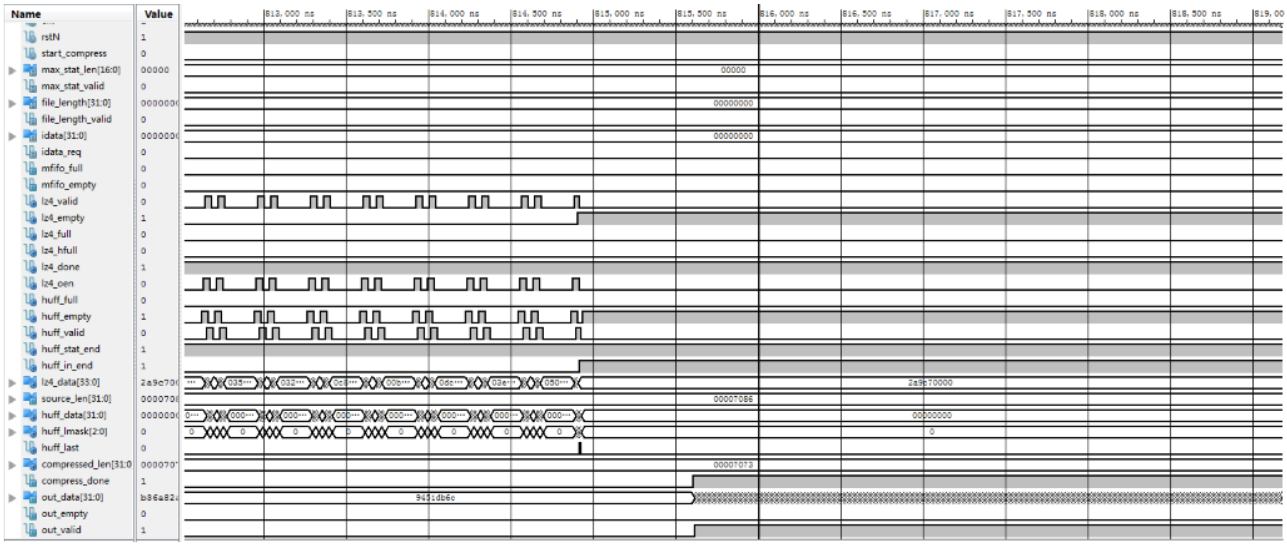


图 4-21 两级压缩电路仿真结束过程波形图

4.5 本章小结

本章提出了一种半静态 Huffman 编码的实现方法及相应的电路结构。针对半静态 Huffman 编码过程中的统计步骤提出了一种基于 RAM 计数器组的并行统计电路，旨在提升统计的速率。针对排序步骤设计了堆排序逻辑和双存储器结构，使电路可以在消耗最少的时钟周期的情况下完成对双存储器中数据的排序。设计了建树电路和编码电路，以状态转移图表示其内部的处理流程。并且，本章为了节约存储空间，控制变长码最大码长，提出了码长优化算法和码表生成算法的实现方法，并设计了对应的电路结构。最后，将本章所述的电路与第三章所述的 LZ4 编码电路进行级联，实现整体电路的兼容模式和优化模式，并实现在 FPGA 上实现了两级压缩电路。