

XVid 源代码剖析

(XVid Version 1.1.0)

主作者: 杨书良

美创算法工作室 www.mcodec.cn

前 言.....	6
第一章 概 述.....	7
1.1 删减判定标准.....	7
1.2 删减的特性.....	7
1.2.1 Sprite 编码.....	7
1.2.2 GMC 编码.....	7
1.2.3 B 帧编码.....	7
1.2.4 complexity_estimation_header.....	8
1.2.5 四分之一像素运动补偿.....	8
1.2.6 reduce 采样.....	8
1.2.7 颜色空间转换.....	8
1.2.8 ASCII 码叠加.....	8
1.2.9 图像后处理.....	8
1.2.10 I 帧 P 帧编码类型判决.....	8
1.2.11 MPEG4 量化.....	8
1.2.12 time 测量.....	8
1.2.13 硬件平台相关.....	8
1.2.14 码率控制.....	9
1.2.15 interlace 编码.....	9
1.2.16 dquant.....	9
1.2.17 Trellis-Based quantization.....	9
1.2.18 4MV 预测.....	9
1.2.19 AC 高级预测.....	9
1.3 修改优化判定标准.....	9
1.4 主要的修改优化.....	9
1.4.1 image_interpolate() 函数的优化.....	9
1.4.2 xvid_me_DiamondSearch() 函数的优化.....	9
1.4.3 xvid_me_SubpelRefine() 函数的优化.....	10
1.4.4 predict_acdc() 函数的优化.....	10
1.4.5 MakeGoodMotionFlags() 函数的优化.....	10
1.4.6 image_input() 函数的优化.....	10
1.4.7 MBQuantInter() 函数的优化.....	10
1.4.8 simplify_time() 函数的优化.....	10
1.4.9 enc_create() 函数的优化.....	10
1.4.10 FrameCodeP() 函数的优化.....	10
1.4.11 bitstream 系列函数的优化.....	10
1.4.12 码率控制系列函数的优化.....	10
1.5 I 帧编码数据流程图.....	10
1.6 P 帧编码流程图.....	12
1.6.1 插值预测帧数据流程图.....	12

1.6.2 判决宏块编码方式流程图.....	12
1.6.3 编码 P 宏块流程图.....	13
1.6.4 不编码 P 宏块流程图.....	14
第二章 主控程序.....	15
2.1 文件列表.....	15
2.2 portab.h 文件.....	15
2.2.1 功能描述.....	15
2.2.2 文件注释.....	15
2.3 global.h 文件.....	16
2.3.1 功能描述.....	16
2.3.2 文件注释.....	16
2.4 encoder.h 文件.....	19
2.4.1 功能描述.....	19
2.4.2 文件注释.....	19
2.5 xvid_encraw.c 文件.....	22
2.5.1 功能描述.....	22
2.5.2 文件注释.....	23
2.6 encoder.c 文件.....	29
2.6.1 功能描述.....	29
2.6.2 文件注释.....	29
第三章 帧级处理程序.....	47
3.1 文件列表.....	47
3.2 image.h 文件.....	47
3.2.1 功能描述.....	47
3.2.2 文件注释.....	47
3.3 image.c 文件.....	47
3.3.1 功能描述.....	47
3.3.2 文件注释.....	49
3.4 interpolate8x8.h 文件.....	60
3.4.1 功能描述.....	60
3.4.2 文件注释.....	60
第四章 运动估计.....	69
4.1 文件列表.....	69
4.2 motion.h 文件.....	69
4.2.1 功能描述.....	69
4.2.2 文件注释.....	69
4.3 sad.h 文件.....	69
4.3.1 功能描述.....	69
4.3.2 文件注释.....	69
4.4 motion_comp.c 文件.....	76

4.4.1 功能描述.....	76
4.4.2 文件注释.....	76
4.5 estimation_pvop.c 文件.....	79
4.5.1 功能描述.....	79
4.5.2 文件注释.....	79
第五章 宏块级实用程序.....	98
5.1 文件列表.....	98
5.2 mem_align.h 文件.....	98
5.2.1 功能描述.....	98
5.2.2 文件注释.....	98
5.3 mem_transfer.h 文件.....	100
5.3.1 功能描述.....	100
5.3.2 文件注释.....	100
5.4 mbfunctions.h 文件.....	109
5.4.1 功能描述.....	109
5.4.2 文件注释.....	109
5.5 mbfunctions.c 文件.....	109
5.5.1 功能描述.....	109
5.5.2 文件注释.....	109
第六章 DCT 变换.....	114
6.1 文件列表.....	114
6.2 fdct.h 文件.....	114
6.2.1 功能描述.....	114
6.2.2 文件注释.....	114
6.3 fdct.c 文件.....	114
6.3.1 功能描述.....	114
6.3.2 文件注释.....	114
6.4 idct.h 文件.....	128
6.4.1 功能描述.....	128
6.4.2 文件注释.....	128
6.5 idct.c 文件.....	128
6.5.1 功能描述.....	128
6.5.2 文件注释.....	128
第七章 量化.....	139
7.1 文件列表.....	139
7.2 quant.h 文件.....	139
7.2.1 功能描述.....	139
7.2.2 文件注释.....	139
7.3 quant_h263.c 文件.....	139
7.3.1 功能描述.....	139

7.3.2 文件注释.....	139
第八章 码流级程序.....	152
8.1 文件列表.....	152
8.2 cbp.h 文件.....	152
8.2.1 功能描述.....	152
8.2.2 文件注释.....	152
8.3 cbp.c 文件.....	152
8.3.1 功能描述.....	152
8.3.2 文件注释.....	152
8.4 vlc_codes.h 文件.....	158
8.4.1 功能描述.....	158
8.4.2 文件注释.....	158
8.5 mbcoding.h 文件.....	175
8.5.1 功能描述.....	175
8.5.2 文件注释.....	175
8.6 mbcoding.c 文件.....	176
8.6.1 功能描述.....	176
8.6.2 文件注释.....	177
8.7 bitstream.h 文件.....	187
8.7.1 功能描述.....	187
8.7.2 文件注释.....	188
8.8 bitstream.c 文件.....	193
8.8.1 功能描述.....	193
8.8.2 文件注释.....	193
附 录.....	197
附录 A 程序用到的部分 MMX/SSE2 汇编指令.....	197
附录 B 备忘录.....	201

前 言

随着国民经济的发展，人们生活水平稳步上升，人们对安全防范的需求越来越多，越来越高，催生了国内数以万计的厂家进入安防行业，整个安防行业最核心的核心技术就是视音频编解码算法。

视音频编解码算法决定了安防产品质量，目前在安防行业视频监控产品按视频压缩方式不同分为芯片硬压缩和 DSP 软压缩。硬压缩方案视频图像压缩质量取决于芯片，选定了主芯片也就决定了最终产品的视频压缩性能，产品开发商没有优化的余地。DSP 软压缩方案需要视频压缩算法，鉴于开发视频压缩算法难度相当大，目前国内只有寥寥几家有实力的公司在做实质性算法开发。

开发 MPEG4 视频编解码算法首先要学习 MPEG4 视频标准，学得差不多的时候就应该跑程序，然后对照程序理解 MPEG4 视频标准，对照 MPEG4 视频标准理解程序，如此不断地循环轮回，算法版的西游记逐步上演，随故事情节展开，历经九九八十一次粹炼，终于有些阶段性成果。本书主作者在开发 MPEG4 视频编码算法时认真研习了 XVid 1.1.0 版，深刻体会没有任何人帮助的条件下一个人阅读理解 MPEG4 视频标准和 XVid 源代码的辛酸苦辣。作者把对 XVid 源代码的理解编写成书，分享给愿意研习 XVid 的工程师们，帮助他们走过那段艰难困苦的历程，缩短他们半年到一年的开发时间。

作者在研习 XVid 源代码的过程中，以嵌入式实际应用为目标环境，拿着斧头砍掉 XVid 中不适应安防要求的的大部分程序，简化了代码量，更容易理解 MPEG4 的精髓，并且触类旁通可以延伸理解几乎所有的 MPEG4 开源代码。经对比测试，简化版图像质量和标准 XVid 1.1.0 基本持平，速度提高 50% 多，给到研习 XVid 的工程师们极大的信心，简单做做减法就有 50% 左右的性能提高，往后优化性能改善更显著，完全优化岂不是性能暴增。事实上本书主作者在看遍 MPEG1/2，MPEG4 视频编解码开源代码后，坚持自主创新，开发的商业级 MPEG4 视频压缩算法和 XVid 相比只损失微小量图像质量却取得 4 倍于 XVid 的超级运算速度，一路 D1 在 Philips TM1502 上轻轻松松跑到实时，商业级的性能，商业级的水准，欢迎各位到 www.mcodec.cn 下载试用。

本书基于作者的简化版来理解 XVid，包括 C 语言版和 PC 汇编版大约 5400 行代码左右，大体按照 XVid 源代码子目录来组织，包括所有文件所有代码，读者也可以到 www.mcodec.cn 下载完整的工程。此源代码在 VC6 打上 vs6sp5 和 vcpp5 补丁后编译通过，因为所有汇编都是嵌入式汇编，故不需要 nasm 编译器。

本书不是一本入门书籍，读者需要理解 MPEG4 ASP 视频标准，熟悉 C 语言和 PC 汇编语言，MMX/SSE2 多媒体加速指令，一些阅读源代码的功底。

主作者：杨书良 

版权说明

美创算法工作室保留本电子书修改和正式出版的所有权利。读者可以自由完整传播本书全部章节的内容，但需要注明出处。由于目前本书还处于草稿阶段，MPEG4 内容深奥宽广，作者水平有限，时间匆忙，书中存在的许多错误和不足，希望读者能踊跃给予批评指正和建议 (tslking@tom.com)，不胜感谢。

版权所有 (C)，2007-2008 美创算法工作室

第一章 概述

1.1 删减判定标准

XVid 的开发平台是 PC 机，各种资源相对很丰富，所以代码量可以做到很庞大；在安防嵌入式产品中，各种资源相对较紧张，每种资源都要节约使用。当把 XVid 从资源相对丰富的 PC 机移到资源相对紧张的嵌入式产品中时，要对 XVid 做一次大规模的瘦身运动，要删掉很大一部分代码。作者以下列几个判定标准来决定是否裁减相关代码。

首先 XVid 的开发目标是图像质量优先，支持很多耗时的特性，用大量的时间来换取图像质量的提高。安防行业是实时性优先，首先要保证编码速度，因此我们在 XVid 的基础上优化的目标是用少量的图像质量换取几倍速度的提高，因此性能时间比不高的特性不会被支持。

其次 XVid 为了和 MPEG4 标准兼容，支持一些实际用途不是很大的特性，并且很耗费 code size，安防行业是嵌入式系统，要保证代码简洁，从而减小 code size，因此有些性能体积比不高的特性不会被支持。

再次 XVid 相对来讲是一个大而全的编码器，尽量满足所有编码需求，而嵌入式编码需求单一，因此有些不符合嵌入式实际需求的特性不会被支持。

最后 XVid 中含有很多的调试代码，嵌入式产品硬件平台和 PC 有根本的不同，调试方法手段有很大差别，所以有些调试特性不会被支持。

按照上面几个评定标准，从程序中删掉相关的代码后，再把代码整理一下，该合并的合并，该移位的移位，这样代码数量减少了一大半，吹尽黄沙始到金，留下的都是精髓。

1.2 删减的特性

1.2.1 Sprite 编码

一个 sprite 是由一个视频段中属于同一个视频对象的所有像素构成的，尤其适用于视频序列的背景编码，在一个全景序列中通过全局运动补偿产生背景 sprite，它包含整个全景序列中所有可见的背景对象像素，在这个背景中的某些部分在某几帧中可能由于前景对象遮挡或相机运动而使得它们不可见。实际中可以用它来直接重构背景的视频对象平面 VOP 或者用于背景 VOP 的预测编码，传送时可以一部分一部分的传送，解码端实时更新。

Sprite 编码目前大多是实验室研究水平，需要很大的技术突破才能满足普适性应用，先删掉。

1.2.2 GMC 编码

GMC 是为了补偿由于摄像机运动、摄像机变焦或者大运动物体引起的全局运动，有助于改善最挑剔的场景中的图像质量，但是需要超级 CPU 超级运算能力。经删减到最后的 XVid 源代码统计，运动估计部分占有大约 30% 左右的 CPU 运算时间，耗时已经很大了，所以以嵌入式 CPU 实际的运算能力来做 GMC 那太为难嵌入式 CPU 了，先删掉。

1.2.3 B 帧编码

因为 B 帧编码时，在编码端，不是按照自然帧顺序编码，帧顺序在内存中要倒来倒去造成时延，同时要占有大量宝贵的嵌入式内存，在解码端，帧顺序也要倒来倒去造成时延，同样要占有大量宝贵的嵌入式内存。在编码和解码两端，既造成时延又消耗内存，先删掉。

1.2.4 complexity_estimation_header

复杂估计头在嵌入式应用中没有太大的作用，还把码流和程序弄得很复杂，是一种花里胡骚的设计，在朴实自然的嵌入式系统中是多余，为简单计，先删掉。

1.2.5 四分之一像素运动补偿

四分之一像素运动补偿比二分之一像素运动补偿更准确，这样运动残差更小，相对就可以减少码流，但是为支持四分之一像素运动补偿需要更大得大的内存空间和更多得多的运动估计时间，在嵌入式属于性能时间比不高和性能空间比不高的特性，先删掉。

1.2.6 reduce 采样

reduce 采样应用于带宽极其不够，把要编码的图像亚采样后，编码更小得图像，解码端通过插值算法还原成原始采样大小，耗费比较大的时间计算来换取一定的灵活性，在嵌入式产品中用不着，先删掉。

1.2.7 颜色空间转换

XVid 用软件转换的方法支持很多种颜色空间，在嵌入式视频应用中一般有图像协处理器硬件来支持颜色空间转换，图像协处理器硬件转换速度更快，因为协处理器硬件可能会做多方向滤波所以图像质量可能更好，所以不用在编码算法中支持颜色空间转换，除 YV12 外其他的全删掉。

1.2.8 ASCII 码叠加

在嵌入式实际应用中是要叠加中文和英文的，但是我们在开始开发算法时不用叠加中文和英文，更好的做法是上一层程序叠加中文和英文，算法只做压缩编码不用支持中英文叠加，先删掉。

1.2.9 图像后处理

图像后处理用于解码算法，通常是用多点滤波来提高解码后的图像质量，在编码器这端不用进行图像后处理，因为 XVid 不仅包括编码，而且还包括解码，编码程序用不着的特性先删掉。

1.2.10 I 帧 P 帧编码类型判决

XVid 通过对每一个宏块计算一大堆 SAD 值来决定当前帧的编码类型是 I 帧还是 P 帧，即自适应 I 帧 P 帧判决。我们现在设定一个固定的 I 帧间隔值来判决编码类型，即 I 帧固定出现在某些位置，比如固定 I 帧间隔 200，即每 200 帧编码一个 I 帧，其余的编码 P 帧。

1.2.11 MPEG4 量化

在 MPEG4 标准中规定可以用 H263 或者 MPEG4 的量化方法对 DCT 变换后系数进行量化，码流里面有字段指示用那种方法，我们以简单实用为原则只支持 H263 的量化方法，这样每个宏块可以减少几次判断，最终减少 CPU 内部的指令流水线中断的次数。

1.2.12 time 测量

做优化的时候对时间的测量很重要，对测量的位置也很敏感，XVid 的方法测量的精度和位置很多不符合实际要求，这部分代码先删掉，在需要的时候再用更好的方法来测量时间。

1.2.13 硬件平台相关

portab.h 文件里面的代码大多是屏蔽 x86 32bit 和 x86 64bit 和 PPC 平台的差别，由于我们目前只支持 PC 机，可以删掉很多很多和硬件有关的代码，并且那些代码中用于 debug 处理的都可以删掉，我们直接用 printf 或断点的方法 debug。

1.2.14 码率控制

XVid 支持多种码率控制方法策略, 比如安防使用的实时编码, PC 机使用的两遍编码等。因为目前我们是实时编码, 因此里面的非实时编码的策略用不上, 只保留 single 策略, 其他的都删掉。

1.2.15 interlace 编码

这个是要做很多计算来判断是否用 interlace 编码, 为减少运算量而把这个删掉。从摄像头过来的图像可能是用 interlace 编码的, 但是在嵌入式系统中, 大多数都有图像协处理器来做 deinterlace, 硬件做又快又好, 实际编码的时候软件就不用考虑 interlace 了, 统一用 progress 编码, 所有和 interlace 相关的代码都删掉。

1.2.16 dquant

码率控制策略可以精确到宏块一级, 即各个宏块用不同的量化系数, 这样有更好的图像质量, 这种方法通常用于非实时两遍压缩, 第一遍计算一些统计参数, 第二遍以第一遍计算的统计参数为指导确定不同的量化系数来得到更好的图像质量。但是我们是实时系统, 不可能做两遍编码, 先删掉。

1.2.17 Trellis-Based quantization

超级吃 CPU 计算能力的家伙, 先删掉。

1.2.18 4MV 预测

为了简化程序代码, 简化逻辑控制策略, 删掉 4MV 预测。

1.2.19 AC 高级预测

为了简化程序代码, 简化逻辑控制策略, 删掉 AC 高级预测, 同时也删掉水平扫描和垂直扫描。

1.3 修改优化判定标准

主要为了减小程序的运算量, 减少判断, 提高运算速度, 把有些计算做了一些简化, 折中平衡一下性能时间比; 把有些计算做了一些变通, 减小运算量, 把有些开关写死, 少做一些比较判断来提速。

1.4 主要的修改优化

1.4.1 image_interpolate() 函数的优化

XVid 原始方法是每次插 8x8 小块, h 和 v 和 hv 三半像素位置分别插值。现改为从左到右, 从上到下, 每次插两行, h 和 v 和 hv 三半像素位置同时插值出来。

1.4.2 xvid_me_DiamondSearch() 函数的优化

经我有限统计表明, 整像素运动估计平均大约 2.5 轮就比较完跳出循环, 因此可以理解为大部分有效整像素运动矢量在三轮之内会找到, 超过三轮还没有找到的宏块多半是因为这个宏块要用 intra 模式编码, 再多的查找都是浪费时间, 所以可以把 for(;;) 循环限定为最多三次, 这样可以避免有些 intra 模式编码的宏块做一些无意义的比较, 还节省时间。

限定最多比较三次后, 就可以把运动矢量范围比较提到循环前面来, 取最大值三次比较, 得到有一个 $6(iDiamondSize*3)$ 的偏差, 用这个大偏差来控制搜索范围, 这样 CheckCandidate 系列函数中的运动矢量范围比较就可以删掉, 用减少比较判断来优化提速。

1.4.3 xvid_me_SubpelRefine() 函数的优化

半像素运动估计最多只差半个像素，所以可以像整像素优化方法一样，把运动矢量范围比较提前，删掉 CheckCandidate 系列函数中的运动矢量范围比较。

根据局部性原理，经我有限统计表明有 95% 最优半像素位置周围是次优位置，因此我们先搜索整像素上下左右共四个半像素点，如果最优点是整像素中心点就不搜索，如果是半像素边缘点再搜最邻近的两个半像素位置，而不是每次都要搜 8 个半像素位置，这样每个点可以少计算两个或四个半像素位置。

1.4.4 predict_acdc() 函数的优化

xvid 里面这个函数对一个宏块的六个小块调用六次，优化一下可以在一个函数里面处理六个小块，此函数只调用一次，不仅可以减小函数调用开销，还减少了很多判断。

1.4.5 MakeGoodMotionFlags() 函数的优化

xvid 用此函数来实现不同的开关选项，但是我们在实际应用中通常会把各个开关选项定死，因此这个函数可以整个删掉。

散落在其他很多函数中的开关判断也可以删掉，减少指令流水线中断次数。

1.4.6 image_input() 函数的优化

由于只支持 YV12 格式的图像，可以把 colorspace 文件删掉，把 image_input() 函数和 yv12_to_yv12_c() 函数合并。

1.4.7 MBQuantInter() 函数的优化

直接删掉耗时的 XVID_VOP_TRELLISQUANT 选项。

通常的实时编码算法不支持 B 帧编码，P 帧的 limit 值为 0，所以我们可以删除很多的比较判断，大大的简化 cbp 的计算方法。

1.4.8 simplify_time() 函数的优化

选择合适的 fincr/fbase，直接删掉此函数

1.4.9 enc_create() 函数的优化

删掉很多不用的缓冲区，节省嵌入式系统宝贵的内存资源。

1.4.10 FrameCodeP() 函数的优化

改变了控制逻辑，思路清晰一些。

1.4.11 bitstream 系列函数的优化

因为 XVid 包括编码和解码部分，我们只需要编码相关的，所以我们可以做很多的简化。大大减小代码量，同时逻辑也清楚明了些。

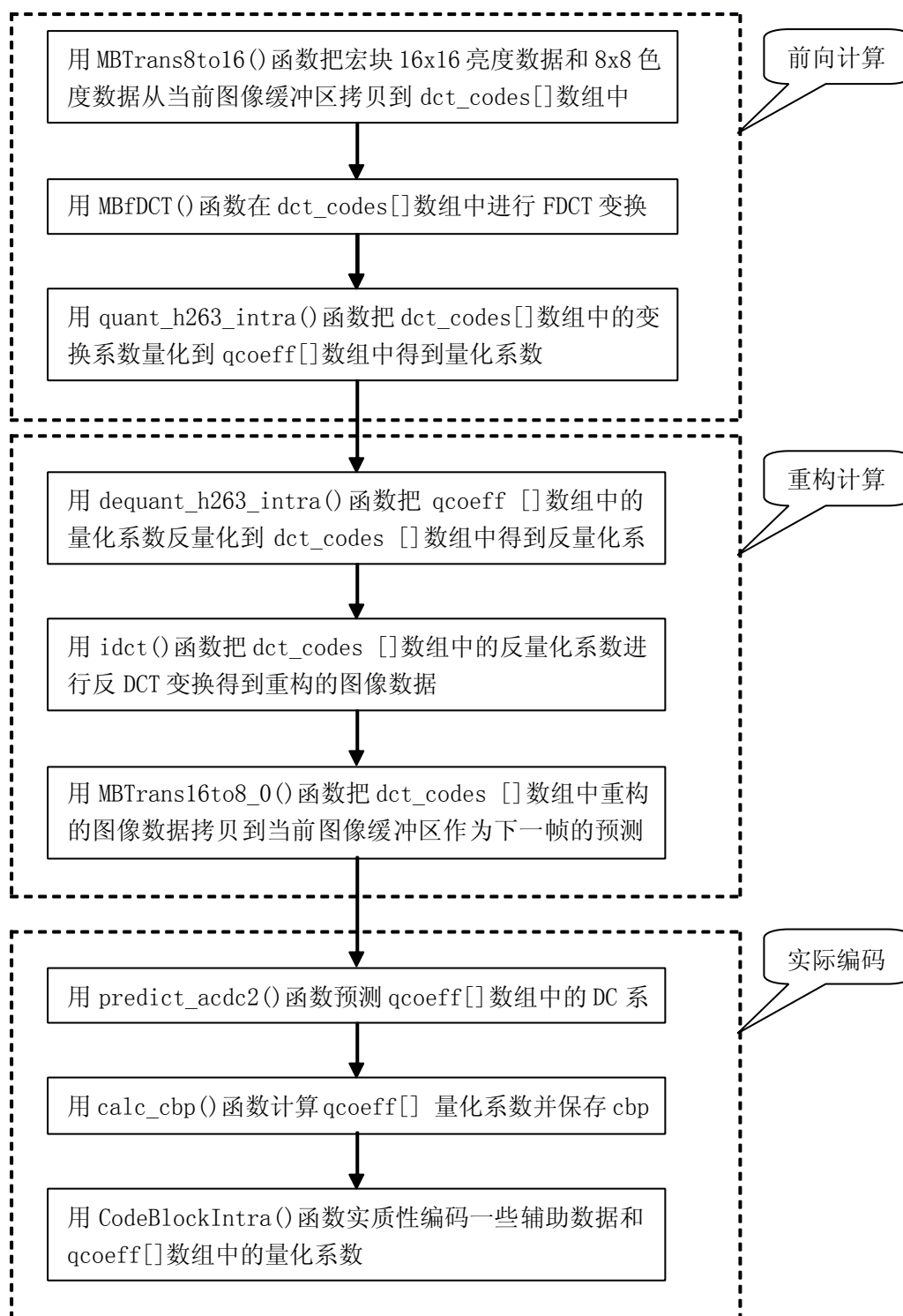
1.4.12 码率控制系列函数的优化

因为我们只支持一种 single 的码率控制策略，在除去相当多的冗余代码后合并到 encoder.c 文件中。

1.5 I 帧编码数据流程图

FrameCodeI() 函数是 I 帧编码总控函数，在声明两个中转数组变量 dct_codes 和 qcoeff 后，设置量化系数，写 I 帧帧头，就进入 I 编码宏块流程图。

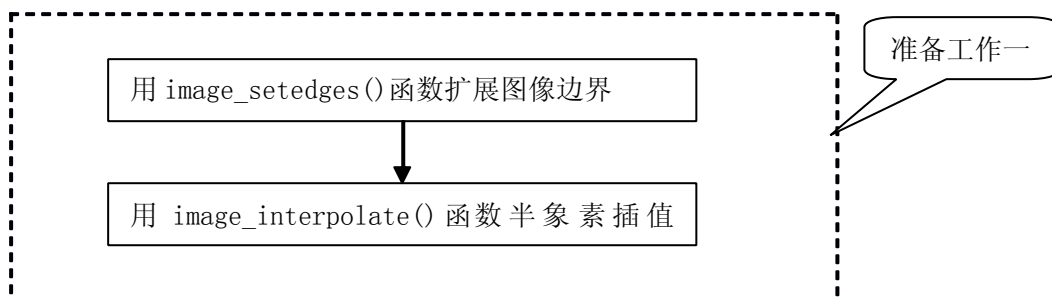
dct_codes 和 qcoeff 数组是关键核心数据区，是编码函数交换计算结果的缓冲区。



图一 I 编码宏块流程图

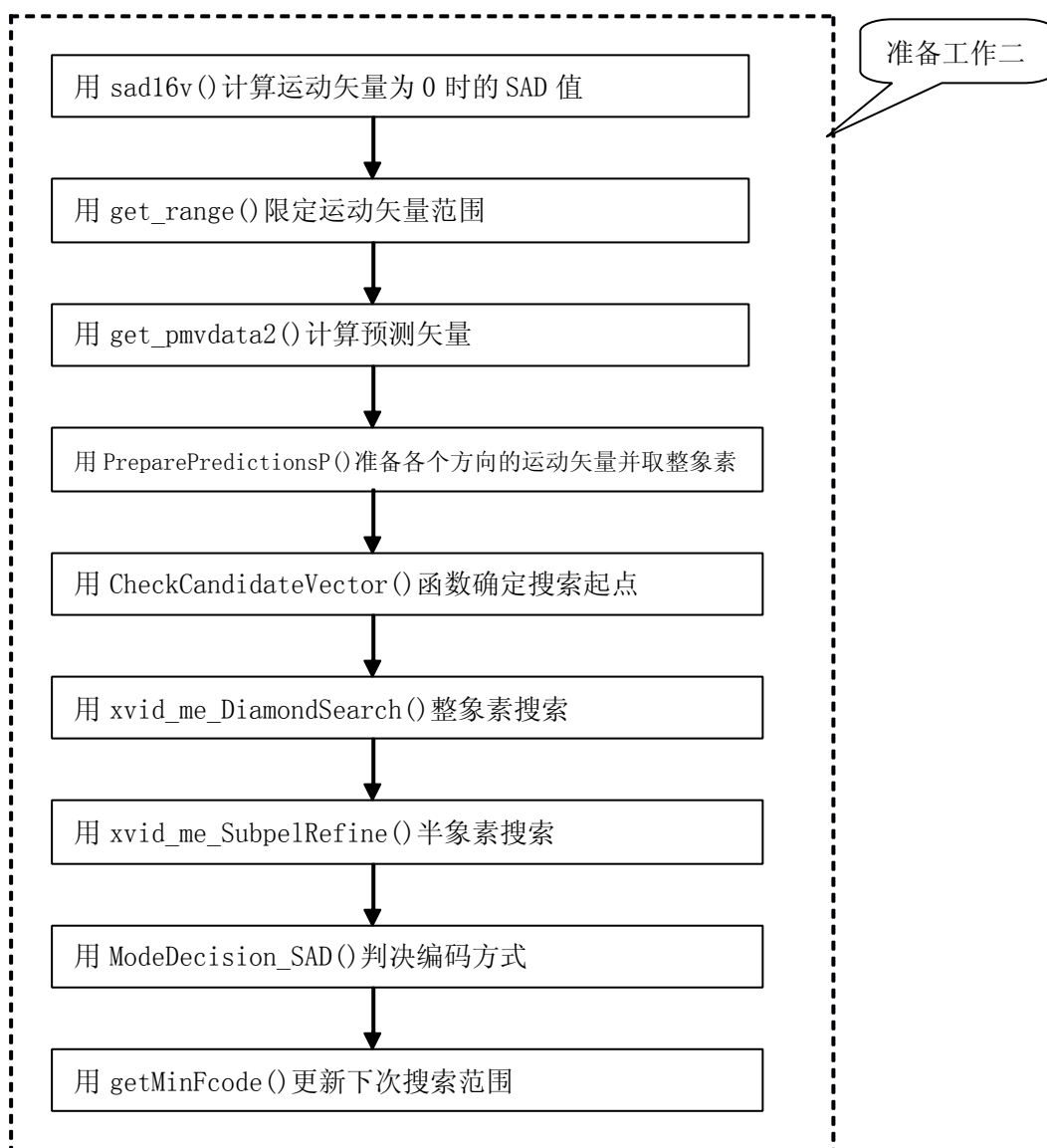
1.6 P 帧编码流程图

1.6.1 插值预测帧数据流程图



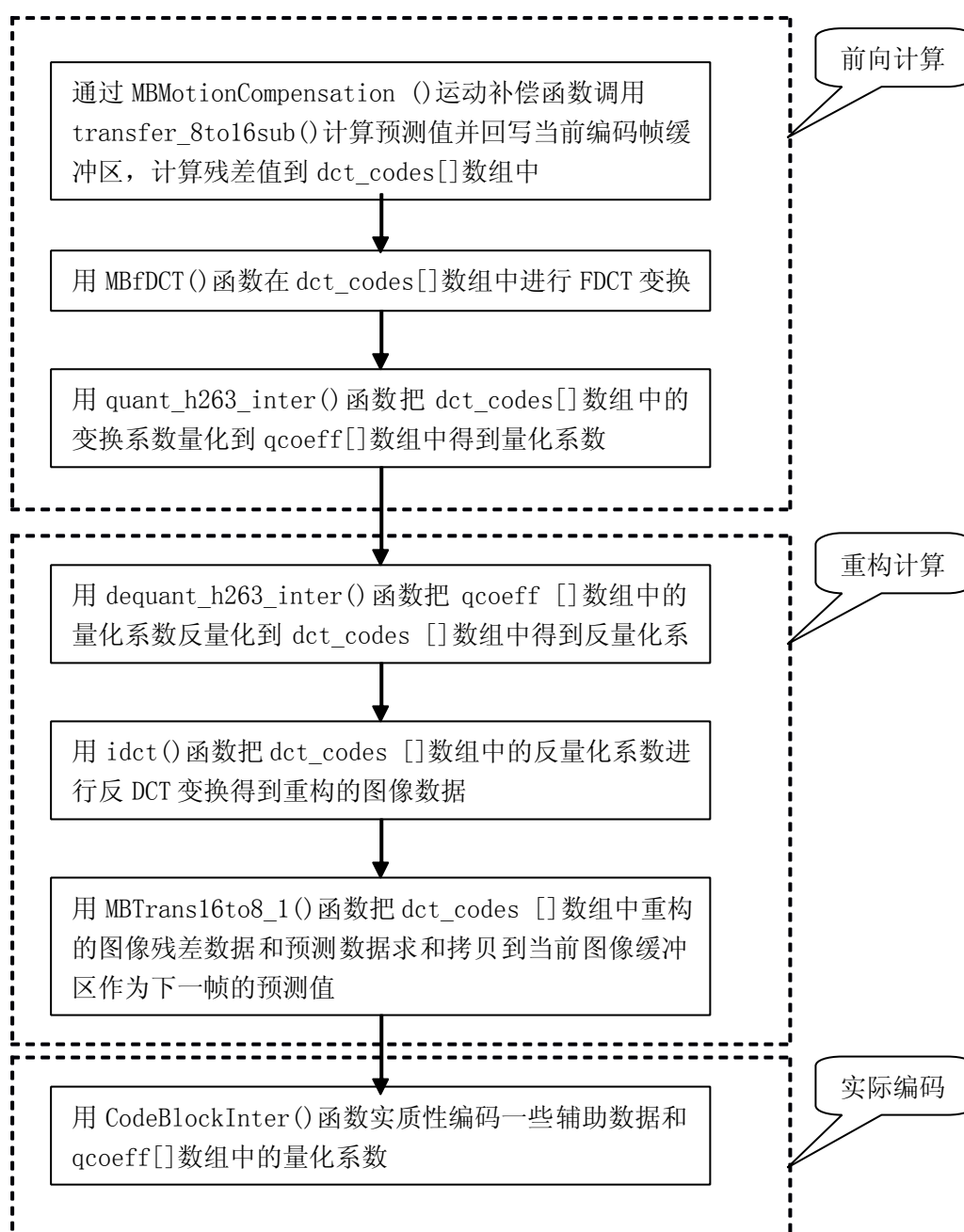
图二 P 帧编码计算预测流程图

1.6.2 判决宏块编码方式流程图



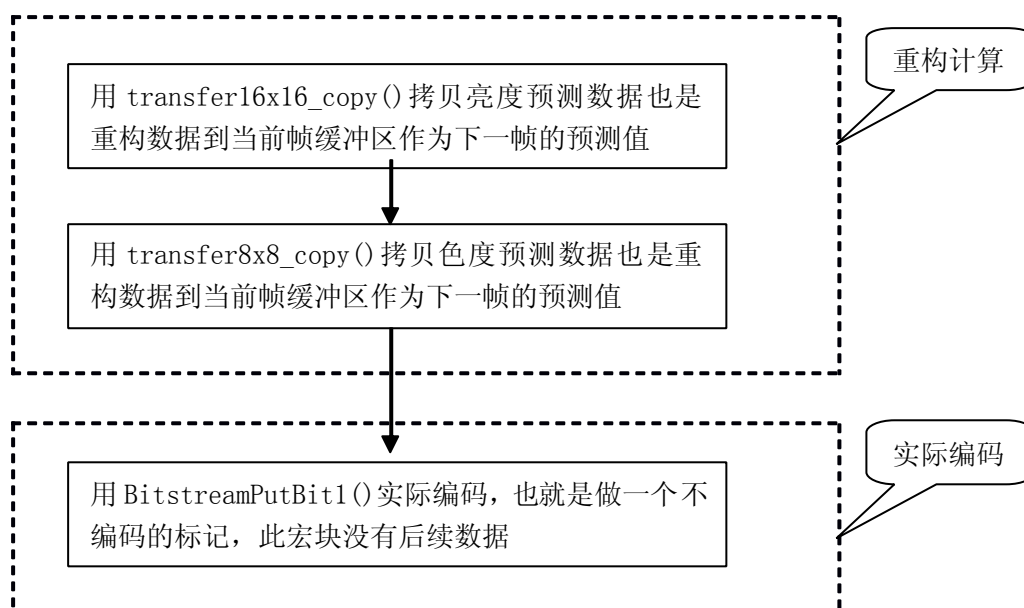
图三 P 帧编码判定宏块编码方式流程图

1.6.3 编码 P 宏块流程图



图四 编码 P 宏块流程图

1.6.4 不编码 P 宏块流程图



图五 不编码 P 宏块流程图

第二章 主控程序

2.1 文件列表

类型	名称	大小
	portab.h	742 bytes
	global.h	1007 bytes
	encoder.h	1289 bytes
	xvid_encraw.c	6043 bytes
	encoder.c	13848 bytes

2.2 portab.h 文件

2.2.1 功能描述

为了方便移植 XVid 到不同的硬件平台上, 该头文件用于定义了一些和硬件平台有关的宏来屏蔽硬件平台的差别, 当前简化的 XVid 只支持 X86 系列 32 位 PC, 所以只需要 PC 硬件平台有关的宏定义, 并且做些简化, 其他硬件平台的宏定义都删掉了。

2.2.2 文件注释

```
1  #ifndef _PORTAB_H_
2  #define _PORTAB_H_

4  #define int8_t    char
5  #define uint8_t   unsigned char
6  #define int16_t   short
7  #define uint16_t  unsigned short
8  #define int32_t   int
9  #define uint32_t  unsigned int
10 #define int64_t   __int64
11 #define uint64_t  unsigned __int64
/* 第 4 行到第 11 行定义一些整个工程有效的全局数据类型, 因为 XVid 要支持不同的硬件平台, 不同的
   编译器, 数据类型表示的数据范围可能会不同, 因此用数据类型宏来屏蔽这些差别。数据类型宏还有一个好处就是相对较短, 程序更简洁。
// */

13 #pragma warning( disable : 4100 4244 4245 4505 4514)
/* 第 13 行是为了屏蔽 VC6 编译时的蹦出来 warning 信息, 这些 warning 信息太多了, disable 一部分后
   更容易找到错误信息。
// */

15 #define CACHE_LINE 64
/* 第 15 行实际是定义分配内存或堆栈时字节对齐的要求, XVid 目前定义 64 字节对齐, 满足 SSE2 汇编
   指令要求。
// */
```

```
17 #define DECLARE_ALIGNED_MATRIX(name, sizex, sizey, type, alignment) \  
18     __declspec(align(alignment)) type name[(sizex)*(sizey)]  
/* 第 17 行和第 18 行定义堆栈分配时字节对齐的宏, XVid 目前定义 64 字节对齐。  
// */  
  
20 #define BSWAP(a) __asm mov eax, a __asm bswap eax __asm mov a, eax  
/* 第 20 行是为了屏蔽数据存储时大端小端的差别, 很简洁的汇编指令  
// */  
  
22 #ifndef NULL  
23 #define NULL (void*)0  
24 #endif  
  
26 #define IDCT_ACC  
27 #define FDCT_ACC  
  
29 #define TRANSFER_ACC  
30 #define QUANT_ACC  
31 #define SAD_ACC  
32 #define INTERPOLATE_ACC  
33 #define INTERPOLATE_ACC  
34 #define CBP_ACC  
/* 第 26 行到第 34 行定义汇编加速开关, 由这些开关指示是否编译汇编加速版本, 这样可以测量每个汇  
   编加速的实际性能, 避开用函数指针时的不大灵活。  
// */  
  
36 #endif
```

2.3 global.h 文件

2.3.1 功能描述

定义整个工程文件有效的并且和硬件平台无关的宏定义和简单数据结构。这些宏和数据结构在编码程序和解码程序中都要用到, 因为我们现在只做编码, 所以只保留和编码有关的宏和数据结构。

XVid 把编码和解码做成一个底层库供上层程序调用, 有些数据结构字段仅仅用于解码程序, 我们把这样的字段也删掉, 简化简化再简化。

2.3.2 文件注释

```
1 #ifndef _GLOBAL_H_  
2 #define _GLOBAL_H_  
  
4 #define MODE_INTER      0  
5 #define MODE_INTER_Q    1  
6 #define MODE_INTER4V    2  
7 #define MODE_INTRA      3
```



```
8  #define MODE_INTRA_Q    4
9  #define MODE_NOT_CODED  16
/* 第 4 行到第 9 行定义宏块类型，目前我们只支持 MODE _INTER, MODE_INTRA, MODE_NOT_CODED 这三种类型，其他类型暂不支持
// */

11 #define I_VOP    0
12 #define P_VOP    1
13 #define B_VOP    2
/* 第 11 行到第 13 行定义 VOP 类型，目前我们只支持 I_VOP, P_VOP 两种类型
// */

15 #define EDGE_SIZE    64
16 #define EDGE_SIZE2   32
17 #define EDGE_SIZE4   16
/* 第 15 行到第 17 行定义扩展边界的大小，Y 分量在原始图像上下左右各扩展 64 像素，U 分量和 V 分量在上下左右各扩展 32 像素。但是注意做半像素插值时 Y 分量上下左右只插值 32 像素，U 分量和 V 分量上下左右只插值 16 像素，限制了运动估计范围没有扩展的图像大。
// */

19 typedef struct
20 {
21     int x;
22     int y;
23 } VECTOR;
/* 第 19 行到第 23 行定义运动矢量结构体，X 方向和 Y 方向的，原点在当前宏块中心，向左 X 方向为负，向右 X 方向为正，向上 Y 方向为负，向下 Y 方向为正。
// */

25 typedef struct
26 {
27     uint8_t *y;
28     uint8_t *u;
29     uint8_t *v;
30 } IMAGE;
/* 第 25 行到第 30 行定义图像结构体，采用 YUV 格式，每个成员指向了一个分量缓冲区。注意 XVid 对图像 Y 分量上下左右做了扩展 64 像素扩展，对 UV 分量上下左右做了 32 像素扩展，但实际使用的扩展像素在上下左右方向只有一半。
// */

32 typedef struct
33 {
34     uint32_t buf;
35     uint32_t pos;
36     uint32_t *tail;
37     uint32_t *start;
38 } Bitstream;
```

```
/* 第 32 行到第 38 行定义码流操作数据结构，
   buf 表示写到码流中的 bit 位数还不够 32 时，暂时拼接这些 bit 位而成的数。
   pos 表示写到码流中的 bit 位数还不够 32 时，计数写了多少个 bit 位。
   tail 表示码流缓冲区末尾指针。
   start 表示码流缓冲区起始指针。
// */

40 typedef struct
41 {
42     VECTOR mvs;

44     short int pred_values[6];

46     int mode;

48     VECTOR pmvs;

50     int32_t sad8[4];
51     int32_t sad16;

53     int cbp;

55 }MACROBLOCK;
/* 第 40 行到第 55 行定义每一个宏块要保存的属性变量。
   mvs 表示当前宏块的运动矢量。
   pred_values 表示 6 个 8×8 小块的 DC 系数值，作为后续 I 块的预测值。
   mode 表示当前宏块的编码模式，MODE_INTER 和 MODE_INTRA 和 MODE_NOT_CODED 这三中模式之一。
   pmvs 表示当前宏块运动矢量减去预测值后的差值。
   sd8[4] 表示 16×16 宏块中 4 个 8×8 小块的 SAD 值。
   sad16 表示 16×16 宏块的 sad 值。
   cbp 表示当前宏块的 cbp 值。
// */

57 #define MIN(X, Y) ((X)<(Y)?(X):(Y))
58 #define MAX(X, Y) ((X)>(Y)?(X):(Y))
/* 第 57 行到第 58 行定义取小值和取大值的宏
// */

60 #define CLIP(X, AMIN, AMAX) (((X)<(AMIN)) ? (AMIN) : ((X)>(AMAX)) ? (AMAX) : (X))
/* 第 60 行定义限幅运算，或者叫做饱和运算
// */

61 #define DIV_DIV(a, b) (((a)>0) ? ((a)+((b)>>1))/(b) : ((a)-((b)>>1))/(b))
/* 第 61 行向上取整除法
// */
```

```
62 #define SWAP(_T_, A, B)    { _T_ tmp = A; A = B; B = tmp; }  
/* 第 62 行定义交换两个变量值的宏  
// */  
  
64 #endif
```

2.4 encoder.h 文件

2.4.1 功能描述

定义编码器使用的一些变量和数据结构，因为编码器需要的数据结构字段比较多，于是 XVid 把这些字段做了一些分类，把相关的字段定义为一个二级数据结构，这样条理清楚很多。

XVid 以图像质量优先支持比较多的高级特性，而我们以实时性优先删掉一些性能时间比或性能空间比不高的高级特性，这样整个数据结构就简单很多，同时为了减少计算和参数传递方便，我们又多加了几个字段。

2.4.2 文件注释

```
1  #ifndef _ENCODER_H_  
2  #define _ENCODER_H_  
  
4  typedef struct  
5  {  
6      uint32_t width;  
7      uint32_t height;  
  
9      uint32_t edged_width;  
10     uint32_t edged_height;  
  
12     uint32_t mb_width;  
13     uint32_t mb_height;  
  
15     int32_t fincr;  
16     uint32_t fbase;  
  
18     uint32_t m_rounding_type;  
19     uint32_t m_fcode;  
  
21     int32_t quant;  
22 } MBParam;  
/* 第 4 行到第 22 行在宏块级定义一些编码参数，主要是 Encoder 结构太大，需要把 Encoder 结构成员按照逻辑关系分大类，这样条例更清晰一些。
```

width, height 以像素为单位表示图像原始宽高;
edged _width, edged_height 以像素为单位表示图像包括扩展边界的宽高
mb_width, mb_height 以宏块为单位表示图像原始宽高;
fincr 表示计算帧率的增量因子, 在简化版中设定为 1;
fbase 表示帧率, 在简化版中定义为 25fps;
m_rounding_type 表示半像素插值时的限定类型, 取值为 0 或 1;
m_fcode 表示运动矢量的一个参数;
quant 表示量化系数;
// */

```
24 typedef struct
25 {
26     int iMvSum;
27     int iMvCount;
28 } Statistics;
/* 第 24 行到第 28 行记录运动矢量参数, 用于更新搜索矢量范围
   iMvSum 表示运动矢量平方和
   iMvCount 表示运动矢量个数
// */
```

```
30 typedef struct
31 {
32     int coding_type;

34     uint32_t y_scale;
35     uint32_t uv_scale;
36     uint32_t mb_width;

38     uint32_t quant;
39     uint32_t rounding_type;
40     uint32_t fcode;

42     IMAGE image;

44     MACROBLOCK *mbs;

46     int length;

48     Statistics sStat;
49 } FRAMEINFO;
```

```
/* 第 30 行到第 50 行定义帧级有效的一些参数。
coding_type 表示编码类型，只支持 I_VOP 和 P_VOP。
y _scale 表示当前帧 Y 分量 DC 缩放系数。
uv_scale 表示当前帧 UV 分量 DC 缩放系数。
mb_width 表示以宏块为单位计算原始图像宽度。
quant 表示当前帧的量化系数。
rounding_type 表示半像素插值时的限定类型，取值为 0 或 1。
fcode 表示运动矢量的一个参数。
image 表示当前帧 YUV 三分量的地址。
mbs 指向当前帧的 MACROBLOCK 集。
length 表示当前帧的长度。
sStat 记录当前帧的状态信息。主要是运动矢量信息，包括运动矢量数目和平方和。
// */

51 typedef struct
52 {
53     int reaction_delay_factor;
54     int averaging_period;
55     int buffer;

57     int bytes_per_sec;
58     double target_framesize;

60     double time;
61     int64_t total_size;
62     int rtn_quant;

64     double sequence_quality;
65     double avg_framesize;
66     double quant_error[31];

68     double fq_error;
69 }rc_single_t;
/* 第 51 行到第 69 行表示 rate control 的一些信息，这部分优化的余地比较小，不用太仔细研究。
// */

71 typedef struct
72 {
73     MBParam mbParam;
```

```
75     int iFrameNum;

77     rc_single_t rc;

79     FRAMEINFO *current;
80     FRAMEINFO *reference;

82     IMAGE image;

84     IMAGE f_refh;
85     IMAGE f_refv;
86     IMAGE f_refhv;

88     float fMvPrevSigma;
89 } Encoder;
/* 第 71 行到第 89 行定义 Encoder 需要的所有变量，因为 Encoder 用到的变量比较多，已经分类，所以
   这里这是一个总集成，各变量大部分在二级结构体中定义。
iFrameNum 用于定义 I 帧间隔。
current 定义当前编码帧。
reference 参考帧。
image 整像素 YUV 分量。
f_refh 水平半像素插值帧。
f_refv 垂直半像素插值帧。
f_refhv 2x2 小方块中心半像素插值帧。
fMvPrevSigma 前一帧的运动矢量平方和。
// */
91 int enc_encode(Encoder * pEnc, uint8_t* inbuf, uint8_t* outbuf,
92               int *key, int32_t stride);

94 #endif
```

2.5 xvid_encraw.c 文件

2.5.1 功能描述

XVid 编码总控程序，因为我们做了开发应用于特殊目的的编码算法，删减了很多的代码，再并入其他文件后，条理更清楚了，主要的功能就是初始化编码器，读 YUV 文件做实际编码，统计一些编码性能信息，最后是释放编码器申请的内存资源。

2.5.2 文件注释

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>

5  #ifndef WIN32
6  #include <sys/time.h>
7  #else
8  #include <time.h>
9  #endif

11 #include "portab.h"
12 #include "global.h"
13 #include "encoder.h"

15 #include "dct/idct.h"
16 #include "utils/mem_align.h"
17 #include "bitstream/bitstream.h"
18 #include "bitstream/mbcoding.h"
19 #include "image/image.h"

21 #define DEFAULT_INITIAL_QUANTIZER 4
22 #define DEFAULT_BITRATE          900000    // 900kbps
23 #define DEFAULT_DELAY_FACTOR     16
24 #define DEFAULT_AVERAGING_PERIOD 100
25 #define DEFAULT_BUFFER           100
/* 第 21 行到第 25 行定义码流控制变量默认值的宏定义，增强程序的可读性
// */

27 static int  ARG_BITRATE = 1536000;
28 static float ARG_FRAMERATE = 25.00f;
29 static char* ARG_INPUTFILE = "d:\\yuv\\yv12_720_480.yuv";
30 static char* ARG_OUTPUTFILE = "D:\\yuv\\enc_720_480.raw";

32 static int XDIM = 720;
33 static int YDIM = 480;
/* 第 27 行到第 33 行写死 XVid 原始命令行基本参数，这样就不用命令行传参数了，虽然不太灵活，但
   收获到很简洁的代码，并且这些变量值基本不变。
// */

35 static Encoder *enc_handle = NULL;

37 #define IMAGE_SIZE(x,y) ((x)*(y)*3/2)
/* 第 37 行定义图像数据帧大小，目前我们只支持 YV12 格式的序列图像集。
// */
```

```
39 static void simplify_time(int *inc, int *base)
40 {
41     int i = *inc;
42     while (i > 1)
43     {
44         if (*inc % i == 0 && *base % i == 0)
45         {
46             *inc /= i;
47             *base /= i;
48             i = *inc;
49             continue;
50         }
51         i--;
52     }
53 }
/* 第 39 行到第 53 行简化时间计算
// */

55 static double msecond()
56 {
57     clock_t clk;

59     clk = clock();
60     return (clk * 1000 / CLOCKS_PER_SEC);
61 }
/* 第 55 行到第 61 行取时间计时，在编码时，时间是一个很重要的评价指标。
// */

63 static int read_yuvdata(FILE * handle, unsigned char *image)
64 {
65     if (fread(image, 1, IMAGE_SIZE(XDIM, YDIM), handle)
66         != (unsigned int) IMAGE_SIZE(XDIM, YDIM))
67         return (1);
68     else
69         return (0);
70 }
/* 第 63 行到第 70 行 从文件读 YUV 数据，一次读完整的一帧数据放到连续的缓冲区中，由后面的
   image_input 来分开 YUV 分量并扩展边界。
// */

72 static int enc_init()
73 {
74     int i;

76     Encoder *pEnc;
```



```
78     pEnc = (Encoder *) xvid_malloc(sizeof(Encoder), CACHE_LINE);
79     memset(pEnc, 0, sizeof(Encoder));
/*     第 78 行到第 79 行申请一块内存来放编码器使用的变量。
//     */

81     enc_handle = pEnc;

83     pEnc->mbParam.width = XDIM;
84     pEnc->mbParam.height = YDIM;
85     pEnc->mbParam.mb_width = (pEnc->mbParam.width + 15) / 16;
86     pEnc->mbParam.mb_height = (pEnc->mbParam.height + 15) / 16;
87     pEnc->mbParam.edged_width = 16 * pEnc->mbParam.mb_width + 2 * EDGE_SIZE;
88     pEnc->mbParam.edged_height = 16 * pEnc->mbParam.mb_height + 2 * EDGE_SIZE;

90     pEnc->mbParam.fincr = 1;
91     pEnc->mbParam.fbase = (int) ARG_FRAMERATE;
92     if (pEnc->mbParam.fincr > 0)
93         simplify_time(&pEnc->mbParam.fincr, &pEnc->mbParam.fbase);
/*     第 83 行到第 93 行编码器基本编码数据赋初值。
//     */

95     pEnc->rc.bytes_per_sec = ARG_BITRATE / 8;
96     pEnc->rc.target_framesize = (double) pEnc->rc.bytes_per_sec
97         / ((double) pEnc->mbParam.fbase / pEnc->mbParam.fincr);

99     pEnc->rc.reaction_delay_factor = DEFAULT_DELAY_FACTOR;
100    pEnc->rc.averaging_period = DEFAULT_AVERAGING_PERIOD;
101    pEnc->rc.buffer = DEFAULT_BUFFER;

103    pEnc->rc.time = 0;
104    pEnc->rc.total_size = 0;
105    pEnc->rc.rtn_quant = 4;

107    for (i = 0; i < 31; i++)
108        pEnc->rc.quant_error[i] = 0.0;

110    pEnc->rc.sequence_quality = 2.0 / (double) pEnc->rc.rtn_quant;
111    pEnc->rc.avg_framesize = pEnc->rc.target_framesize;

113    pEnc->rc.fq_error = 0;
/*     第 95 行到第 113 行是给编码器与码流控制有关的变量的初始值。
//     */

115    pEnc->current = xvid_malloc(sizeof(FRAMEINFO), CACHE_LINE);
116    pEnc->reference = xvid_malloc(sizeof(FRAMEINFO), CACHE_LINE);
```

```
118     pEnc->current->mbs = xvid_malloc(sizeof(MACROBLOCK) * pEnc->mbParam.mb_width *
119                                     pEnc->mbParam.mb_height, CACHE_LINE);

121     pEnc->reference->mbs = xvid_malloc(sizeof(MACROBLOCK) * pEnc->mbParam.mb_width *
122                                     pEnc->mbParam.mb_height, CACHE_LINE);

124     image_create(&pEnc->f_refh, pEnc->mbParam.edged_width, pEnc->mbParam.edged_height);
125     image_create(&pEnc->f_refv, pEnc->mbParam.edged_width, pEnc->mbParam.edged_height);
126     image_create(&pEnc->f_refhv, pEnc->mbParam.edged_width, pEnc->mbParam.edged_height);

128     image_create(&pEnc->current->image, pEnc->mbParam.edged_width,
129                 pEnc->mbParam.edged_height);
130     image_create(&pEnc->reference->image, pEnc->mbParam.edged_width,
131                 pEnc->mbParam.edged_height);

133     image_create(&pEnc->image, pEnc->mbParam.edged_width, pEnc->mbParam.edged_height);
/*     第 115 行到第 133 行申请一些内存来保存中间变量。
//     */

135     pEnc->iFrameNum = 0;
136     pEnc->fMvPrevSigma = -1;

138     idct_int32_init(); // 初始化 idct 限幅表, 如果用 MMX/SSE2 等指令做 idct 就不用这个表
139     init_vlc_tables(); // 初始化 VLC 表, 用空间换时间的做法。

141     return 1;
142 }

144 static int enc_stop()
145 {
146     Encoder * pEnc = (Encoder*)enc_handle;

148     image_destroy(&pEnc->image, pEnc->mbParam.edged_width);

150     image_destroy(&pEnc->current->image, pEnc->mbParam.edged_width);
151     image_destroy(&pEnc->reference->image, pEnc->mbParam.edged_width);

153     image_destroy(&pEnc->f_refh, pEnc->mbParam.edged_width);
154     image_destroy(&pEnc->f_refv, pEnc->mbParam.edged_width);
155     image_destroy(&pEnc->f_refhv, pEnc->mbParam.edged_width);

157     xvid_free(pEnc->current->mbs);
158     xvid_free(pEnc->current);

160     xvid_free(pEnc->reference->mbs);
161     xvid_free(pEnc->reference);
```

```
163     xvid_free(pEnc);

165     return 1;
166 }
/* 第 144 行到第 166 行释放编码器开始申请的内存。
// */

168 int main(int argc, char *argv[])
169 {
170     unsigned char *mp4_buffer = NULL;
171     unsigned char *in_buffer = NULL;

173     double enctime=0;

175     unsigned int toltime=0;

177     int m4v_size=0;
178     int key=0;

180     int result = 0;
181     int input_num = 0;

183     FILE *in_file = stdin;
184     FILE *out_file = NULL;

186     in_file = fopen(ARG_INPUTFILE, "rb");
187     if (in_file == NULL)
188     {
189         fprintf(stderr, "Error opening input file %s\n", ARG_INPUTFILE);
190         return (-1);
191     }

193     if ((out_file = fopen(ARG_OUTPUTFILE, "w+b")) == NULL)
194     {
195         fclose(in_file);
196         fprintf(stderr, "Error opening output file %s\n", ARG_OUTPUTFILE);
197         return (-1);
198     }

200     in_buffer = (unsigned char *) malloc(IMAGE_SIZE(XDIM, YDIM));
201     if (!in_buffer)
202     {
203         fclose(in_file);
204         fclose(out_file);
205         return (-1);
```

```
206     }
/*      第 200 行到第 206 行动态分配输入缓冲区，只需要能完整放一帧数据就好。
//      */

208     mp4_buffer = (unsigned char *) malloc(IMAGE_SIZE(XDIM, YDIM) * 2);
209     if (!mp4_buffer)
210     {
211         fclose(in_file);
212         fclose(out_file);
213         free(in_buffer);
214         return (-1);
215     }
/*      第 208 行到第 215 行动态分配输出缓冲区，通常这个缓冲区为保险起见都分配的比较大，通常大
      于输入缓冲区。
//      */

217     enc_init();

219     do
220     {
221         if (!result)
222             result = read_yuvdata(in_file, in_buffer);

224         if (result)
225             break;

227         enc_time = msecond();
228         m4v_size = enc_encode((Encoder *)enc_handle, in_buffer, mp4_buffer, &key, XDIM);
229         enc_time = msecond() - enc_time;

231         printf("frame=%4d, key=%i, time=%4.0f, quant=%d, len=%5d \n",
232             input_num, key, (float)enc_time, enc_handle->current->quant, (int)m4v_size);

234         if (m4v_size < 0)
235             break;

237         if (m4v_size > 0)
238             fwrite(mp4_buffer, 1, m4v_size, out_file);

240         input_num++;

242         tot_time+=enc_time;

244     } while (input_num<=1000);
/*      第 219 行到第 244 行是编码程序主循环，流程就是读一帧 YUV 数据，编码一帧 YUV 数据，把编码
      好的码流写到输出文件，更新帧计数和总时间。
```

```
//      */

246     printf("time=%d \n", tolttime);

248     enc_stop();

250     free(mp4_buffer);
251     free(in_buffer);

253     fclose(in_file);
254     fclose(out_file);
/*      第 248 行到第 254 行是编码程序编码完后，释放申请的资源。
//      */

256     return (0);
257 }
```

2.6 encoder.c 文件

2.6.1 功能描述

XVid 编码器帧级编码相关的程序，主要是码率控制中的量化系数的计算，然后调用编码函数转入 I 帧和 P 帧编码函数，再在 I 帧和 P 帧编码函数中转入宏块一级的编码函数完成实际的编码。

2.6.2 文件注释

```
1  #include <stdlib.h>
2  #include <math.h>

4  #include "portab.h"
5  #include "global.h"
6  #include "encoder.h"

8  #include "bitstream/bitstream.h"

10 #include "image/image.h"
11 #include "motion/motion.h"

13 #include "utils/mbfunctions.h"
14 #include "bitstream/mbcoding.h"
15 #include "dct/idct.h"
16 #include "utils/mem_transfer.h"
17 #include "bitstream/cbp.h"
```

```
19 static int FrameCodeI(Encoder * pEnc, Bitstream * bs);
20 static int FrameCodeP(Encoder * pEnc, Bitstream * bs);

22 static __inline uint32_t get_dc_scaler(uint32_t quant, uint32_t lum)
23 {
24     if (quant < 5)
25         return 8;

27     if (quant < 25 && !lum)
28         return (quant + 13) / 2;

30     if (quant < 9)
31         return 2 * quant;

33     if (quant < 25)
34         return quant + 8;

36     if (lum)
37         return 2 * quant - 16;
38     else
39         return quant - 6;
40 }

/* 第 22 行到第 40 行完全表达了 MP4 标准规定的 DC 系数和 quant_scale 的关系表
```

DC 量化值同 quantiser scale 之间的关系表				
Component:Type	Dc scaler for quantiser scale range			
	1 through 4	5 through 8	9 through 24	>= 25
Luminance: Type1	8	2x quantiser scale	quantiser scale +8	2 x quantiser scale -16
Chrominance: Type2	8	(quantiser scale +13)/2		quantiser scale -6

```
// */
```

```
42 static void call_plugins_0(Encoder * pEnc, int * quant)
43 {
44     int q = pEnc->rc.rtn_quant;

46     if (q > 31)    q = 31;
47     else if (q < 4) q = 4;

49     pEnc->current->quant = q;
```

```
50     pEnc->mbParam.quant = q;
51 }
/* 第 42 行到第 51 行是从码率控制函数中修改而来，每一帧编码前调用此函数取到码率控制函数计算出来的 quant，做限幅运算限制在[4, 31]之间。我们认为 quant 在 4 以下图像质量会非常好，在实际情况下出现这种情况时把 quant 限制在 4，图像质量影响不会非常大，节省一些 bit 出来分配到以后帧中，这样提高整个图像序列的整体质量，避免前后帧 quant 值差别太远图像质量差别太大。
// */

53 static void call_plugins_1(Encoder * pEnc, FRAMEINFO * frame)
54 {
55     int64_t deviation;
56     int rtn_quant;
57     double overflow;
58     double averaging_period;
59     double reaction_delay_factor;
60     double quality_scale;
61     double base_quality;
62     double target_quality;

64     rc_single_t *rc = &(pEnc->rc);

66     rc->time += (double) pEnc->mbParam.fincr / pEnc->mbParam.fbase;
67     rc->total_size += frame->length;

69     deviation = (__int64)(rc->total_size - rc->bytes_per_sec * rc->time);

71     averaging_period = (double) rc->averaging_period;

73     rc->sequence_quality -= rc->sequence_quality / averaging_period;

75     rc->sequence_quality += 2.0 / (double) frame->quant / averaging_period;

77     if (rc->sequence_quality < 0.1)
78         rc->sequence_quality = 0.1;
79     else if (rc->sequence_quality > 1.0)
80         rc->sequence_quality = 1.0;

82     if (frame->coding_type != I_VOP)
83     {
84         reaction_delay_factor = (double) rc->reaction_delay_factor;
```

```
85         rc->avg_framesize -= rc->avg_framesize / reaction_delay_factor;
86         rc->avg_framesize += frame->length / reaction_delay_factor;
87     }

89     if (frame->coding_type == B_VOP)
90         return ;

92     quality_scale = rc->target_framesize / rc->avg_framesize
93                   * rc->target_framesize / rc->avg_framesize;

95     base_quality = rc->sequence_quality;
96     if (quality_scale >= 1.0)
97         base_quality = 1.0 - (1.0 - base_quality) / quality_scale;
98     else
99         base_quality = 0.06452 + (base_quality - 0.06452) * quality_scale;

101     overflow = -((double) deviation / (double) rc->buffer);

103     if (overflow > rc->target_framesize)
104         overflow = rc->target_framesize;
105     else if (overflow < -rc->target_framesize)
106         overflow = -rc->target_framesize;

108     target_quality = base_quality
109                   + (base_quality - 0.06452) * overflow / rc->target_framesize;

111     if (target_quality > 2.0)
112         target_quality = 2.0;
113     else if (target_quality < 0.06452)
114         target_quality = 0.06452;

116     rtn_quant = (int) (2.0 / target_quality);

118     if (rtn_quant > 0 && rtn_quant < 31)
119     {
120         rc->quant_error[rtn_quant - 1] += 2.0 / target_quality - rtn_quant;
121         if (rc->quant_error[rtn_quant - 1] >= 1.0)
122         {
123             rc->quant_error[rtn_quant - 1] -= 1.0;
124             rtn_quant++;

```



```
125         rc->rtn_quant++;
126     }
127 }

129     if (rtn_quant > rc->rtn_quant + 1)
130     {
131         if (rtn_quant > rc->rtn_quant + 3)
132             if (rtn_quant > rc->rtn_quant + 5)
133                 rtn_quant = rc->rtn_quant + 3;
134             else
135                 rtn_quant = rc->rtn_quant + 2;
136         else
137             rtn_quant = rc->rtn_quant + 1;
138     }
139     else if (rtn_quant < rc->rtn_quant - 1)
140     {
141         if (rtn_quant < rc->rtn_quant - 3)
142             if (rtn_quant < rc->rtn_quant - 5)
143                 rtn_quant = rc->rtn_quant - 3;
144             else
145                 rtn_quant = rc->rtn_quant - 2;
146         else
147             rtn_quant = rc->rtn_quant - 1;
148     }

150     rc->rtn_quant = rtn_quant;
151 }
/* 第 53 行到第 151 行是每一帧编码完后码率控制函数使用一定的策略来计算下一帧要使用的 quant 值。
// */

153 int enc_encode(Encoder* pEnc, uint8_t* inbuf, uint8_t* outbuf, int* key, int32_t stride)
154 {
155     Bitstream bs;

157     BitstreamInit(&bs, outbuf, 0);

159     image_input(&pEnc->image, pEnc->mbParam.width, pEnc->mbParam.height,
160                pEnc->mbParam.edged_width, inbuf, stride);

162     SWAP(FRAMEINFO*, pEnc->current, pEnc->reference);
```

```
164     image_swap(&pEnc->current->image, &pEnc->image);

166     pEnc->current->fcode = pEnc->mbParam.m_fcode;

168     call_plugins_0(pEnc, &pEnc->current->quant);

170     pEnc->iFrameNum++;

172     if(pEnc->iFrameNum>=0)
173     {
174         pEnc->iFrameNum=-200;

176         FrameCodeI(pEnc, &bs);    // I 帧编码总控函数
177         *key = 2;                  // 返回关键帧标记
178     }
179     else
180     {
181         FrameCodeP(pEnc, &bs);    // P 帧编码总控函数
182         *key = 0;
183     }

185     call_plugins_1(pEnc, pEnc->current);

187     return BitstreamLength(&bs); // 当前编码帧长度
188 }
```

/* 第 153 行到第 188 行是编码函数, 首先初始化输出缓冲区, 也就是复位输出缓冲区。接着用 image_input 函数把 YUV 数据分别拷贝到 pEnc->image 中, 并且留出边界扩展的空间。

第 162 行到第 164 行是交换一下当前编码帧和参考帧的有关数据域的指针, 因为我们要支持 I_VOP 和 P_VOP, 每编码完一帧后开始下一个循环时, 丢弃原来的预测帧, 把当前已编码完的那一帧做为预测帧, 新读进原始 YUV 数据的那一帧作为当前编码帧, 交换指针比直接拷贝数据效率要高很多。

第 168 行是取得当前帧的 quant, 这个值是码率控制函数按照一定的策略计算出来的。

第 170 行用 iFrameNum 来计数 I 帧间隔, 判定编码帧类型, 然后进行相应帧编码。

第 185 行是调用码率控制函数来计算下一帧的 quant

第 187 行返回当前帧占用了多少个字节。

// */

```
190 static __inline void CodeIntraMB(Encoder * pEnc, MACROBLOCK * pMB)
191 {
192     pMB->mode = MODE_INTRA;
```

```
194     pMB->mvs.x = pMB->mvs.y = 0;
195     pMB->sad8[0] = pMB->sad8[1] = pMB->sad8[2] = pMB->sad8[3] = 0;
196     pMB->sad16 = 0;
197 }
/* 第 190 行到第 197 行是复位 intra macroblock 的记录信息，用于当前帧和下一帧预测
// */

199 static const int16_t default_acdc_values[1] = {1024};
/* 因为我们只支持 DC 系数预测，只需记录 DC 系数，不用记录 AC 系数，所以数组只有一项。
// */

201 static void predict_acdc2(MACROBLOCK * pMB, FRAMEINFO *frame, const int32_t x,
202                          const int32_t y, int16_t *dctcoeff)
203 {
204     const int16_t *pLeft = default_acdc_values;
205     const int16_t *pTop = default_acdc_values;
206     const int16_t *pDiag = default_acdc_values;

208     const int16_t mb_width=frame->mb_width;

210     int32_t iDcScaler=frame->y_scale;
211     int32_t iDcScaler2 = iDcScaler >> 1;

213     int16_t predictors;
214     int16_t *pCurDC;

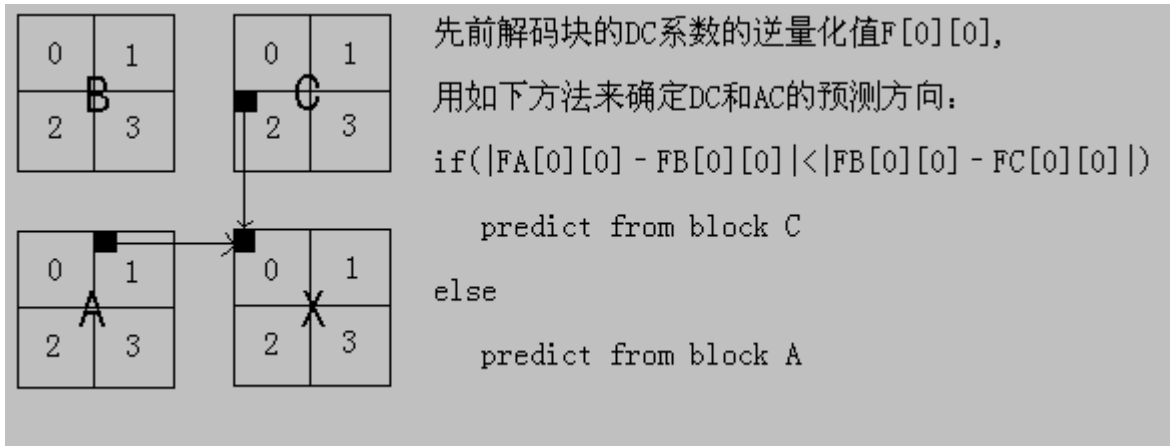
216     int16_t *left, *top, *diag, *current;

218     left = top = diag = current = 0;

220     if (x && pMB[-1].mode == MODE_INTRA ) // 左边
221         left = pMB[-1].pred_values;

223     if (y && pMB[-mb_width].mode == MODE_INTRA ) // 上边
224         top = pMB[-mb_width].pred_values;

226     if (x && y && pMB [-1 - mb_width].mode == MODE_INTRA ) // 左上边
227         diag = pMB[- 1 - mb_width].pred_values;
/*
```



图六 Intra 宏块 DC 系数预测示意图

第 222 行到第 131 行确定宏块左边, 上边, 左上边的预测值指针, 预测 X 宏块 DC 系数示意图如上。

第 221 行相当于 left 指向 A 宏块的 DC 系数预测值数组。

第 224 行相当于 top 指向 C 宏块的 DC 系数预测值数组。

第 227 行相当于 diag 指向 B 宏块的 DC 系数预测值数组。

注意 pred_values 数组只有六个元素, 四个亮度预测值, 两个色度预测值。

```
// */
```

```
229     current = pMB->pred_values;
```

```
231     if (left)    // block = 0
```

```
232         pLeft = left + 1;
```

```
234     if (top)
```

```
235         pTop = top + 2;
```

```
237     if (diag)
```

```
238         pDiag = diag + 3;
```

/* 第 229 行到第 238 行计算预测宏块 X 的第 0 个 8x8 小块的候选分量, 按照 MPEG4 标准规定, 相关的预测值可能由 A 宏块的第 1 小块 DC 值, C 宏块的第 2 小块 DC 值, B 宏块的第 3 小块 DC 值计算而来, 所以有第 232 行第 235 行第 238 行代码分别加 1, 2, 3 的偏移计算, 注意从 0 开始计数。

```
// */
```

```
240     if (abs(pLeft[0] - pDiag[0]) < abs(pDiag[0] - pTop[0]))
```

```
241         predictors = (pTop[0] + iDcScaler2) / iDcScaler;
```

```
242     else
```

```
243         predictors = (pLeft[0] + iDcScaler2) / iDcScaler;
```

/* 第 240 行到第 243 行 计算 DC 自适应预测值

```
// */
```

```
245     pCurDC = pMB->pred_values;
246     *pCurDC = dctcoeff[0] * iDcScaler;
247     *pCurDC = CLIP(*pCurDC, -2048, 2047);

249     dctcoeff[0] = dctcoeff[0] - predictors;
/*     第 245 行到第 247 行规整计算 DC 预测值并保存在 pred_values[0]中。
     第 249 行 计算 DC 差值。
//     */

252     pLeft = current; // block = 1

254     if (top)
255     {
256         pTop = top + 3;
257         pDiag = top + 2;
258     }
259     else
260     {
261         pTop = default_acdc_values;
262         pDiag = default_acdc_values;
263     }
/*     第 252 行到第 263 行计算预测宏块 X 的第一个 8x8 小块的候选分量, 按照 MPEG4 标准规定, 相关的
     预测值可能由当前 X 宏块的第 0 小块 DC 值, C 宏块的第 2 小块 DC 值和第 3 小块 DC 值计算而
     来, 所以有第 256 行第 257 行代码分别加 2, 3 的偏移计算, 注意从 0 开始计数。
//     */
265     if (abs(pLeft[0] - pDiag[0]) < abs(pDiag[0] - pTop[0]))
266         predictors = (pTop[0] + iDcScaler2) / iDcScaler;
267     else
268         predictors = (pLeft[0] + iDcScaler2) / iDcScaler;

270     pCurDC = pMB->pred_values+1; // 规整计算 DC 预测值并保存在 pred_values[1]中。
271     *pCurDC = dctcoeff[64] * iDcScaler;
272     *pCurDC = CLIP(*pCurDC, -2048, 2047);

274     dctcoeff[64] = dctcoeff[64] - predictors;

277     pTop = current; // block = 2

279     if (left)
280     {
```

```
281     pLeft = left + 3;
282     pDiag = left + 1;
283 }
284 else
285 {
286     pLeft = default_acdc_values;
287     pDiag = default_acdc_values;
288 }
/* 第 277 行到第 288 行计算预测宏块 X 的第二个 8x8 小块的候选分量, 按照 MPEG4 标准规定, 相关的
   预测值可能由当前 X 宏块的第 0 小块 DC 值, A 宏块的第 1 小块 DC 值和第 3 小块 DC 值计算而
   来, 所以有第 281 行第 282 行代码分别加 1, 3 的偏移计算, 注意从 0 开始计数。
// */

290 if (abs(pLeft[0] - pDiag[0]) < abs(pDiag[0] - pTop[0]))
291     predictors = (pTop[0] + iDcScaler2) / iDcScaler;
292 else
293     predictors = (pLeft[0] + iDcScaler2) / iDcScaler;

295 pCurDC = pMB->pred_values+2; // 规整计算 DC 预测值并保存在 pred_values[2] 中。
296 *pCurDC = dctcoeff[128] * iDcScaler;
297 *pCurDC = CLIP(*pCurDC, -2048, 2047);

299 dctcoeff[128] = dctcoeff[128] - predictors;

302 pLeft = current + 2;    // block = 3
303 pTop = current + 1;
304 pDiag = current;
/* 第 302 行到第 304 行计算预测宏块 X 的第三个 8x8 小块的候选分量, 按照 MPEG4 标准规定, 相关的
   预测值由当前 X 宏块的第 0 小块 DC 值第 1 小块 DC 值和第 2 小块 DC 值计算而来, 所以有第
   302 行和第 303 行代码分别加 2, 1 的偏移计算, 注意从 0 开始计数。
// */

306 if (abs(pLeft[0] - pDiag[0]) < abs(pDiag[0] - pTop[0]))
307     predictors = (pTop[0] + iDcScaler2) / iDcScaler;
308 else
309     predictors = (pLeft[0] + iDcScaler2) / iDcScaler;

311 pCurDC = pMB->pred_values+3; // 规整计算 DC 预测值并保存在 pred_values[3] 中。
312 *pCurDC = dctcoeff[192] * iDcScaler;
```

```
313     *pCurDC = CLIP(*pCurDC, -2048, 2047);

315     dctcoeff[192] = dctcoeff[192] - predictors;

318     iDcScaler=frame->uv_scale;
319     iDcScaler2 = iDcScaler >> 1;

321     if (left)    // block = 4
322         pLeft = left + 4;
323     else
324         pLeft = default_acdc_values;
325     if (top)
326         pTop = top + 4;
327     else
328         pTop = default_acdc_values;
329     if (diag)
330         pDiag = diag + 4;
331     else
332         pDiag = default_acdc_values;
/* 第 321 行到第 332 行计算预测宏块 X 的第四个 8x8 小块的候选分量, 此小块是色度块, 按照 MPEG4
   标准规定, 相关的预测值可能是相邻宏块对应的色度 DC 值计算而来。所以有第 322 行和第 326
   行和第 330 行代码分别加 4 的偏移计算, 注意从 0 开始计数。
// */

334     if (abs(pLeft[0] - pDiag[0]) < abs(pDiag[0] - pTop[0]))
335         predictors = (pTop[0] + iDcScaler2) / iDcScaler;
336     else
337         predictors = (pLeft[0] + iDcScaler2) / iDcScaler;

339     pCurDC = pMB->pred_values+4; // 规整计算 DC 预测值并保存在 pred_values[4]中。
340     *pCurDC = dctcoeff[256] * iDcScaler;
341     *pCurDC = CLIP(*pCurDC, -2048, 2047);

343     dctcoeff[256] = dctcoeff[256] - predictors;

345     if (left)    // block = 5
346         pLeft = left + 5;
347     else
348         pLeft = default_acdc_values;
```

```
349     if (top)
350         pTop = top + 5;
351     else
352         pTop = default_acdc_values;
353     if (diag)
354         pDiag = diag + 5;
355     else
356         pDiag = default_acdc_values;
/* 第 345 行到第 356 行计算预测宏块 X 的第 5 个 8x8 小块的候选分量, 此小块是色度块, 按照 MPEG4
   标准规定, 相关的预测值可能是相邻宏块对应的色度 DC 值计算而来。所以有第 346 行第 350
   行第 354 行代码分别加 5 的偏移计算, 注意从 0 开始计数。
// */
358     if (abs(pLeft[0] - pDiag[0]) < abs(pDiag[0] - pTop[0]))
359         predictors = (pTop[0] + iDcScaler2) / iDcScaler;
360     else
361         predictors = (pLeft[0] + iDcScaler2) / iDcScaler;

363     pCurDC = pMB->pred_values+5; // 规整计算 DC 预测值并保存在 pred_values[5]中。
364     *pCurDC = dctcoeff[320] * iDcScaler;
365     *pCurDC = CLIP(*pCurDC, -2048, 2047);

367     dctcoeff[320] = dctcoeff[320] - predictors;
368 }

370 static int FrameCodeI(Encoder * pEnc, Bitstream * bs) // I 帧编码总控函数
371 {
372     int bits = BitstreamPos(bs);
373     int mb_width = pEnc->mbParam.mb_width;
374     int mb_height = pEnc->mbParam.mb_height;

376     DECLARE_ALIGNED_MATRIX(dct_codes, 6, 64, int16_t, CACHE_LINE);
377     DECLARE_ALIGNED_MATRIX(qcoeff, 6, 64, int16_t, CACHE_LINE);
/* 第 376 行到第 377 行声明了 I 帧编码过程中的核心交换数据区, 以此数据区为中心, 调用相关的
   过程函数按照 MP4 标准分步骤来处理, 多跟踪几次这两个核心交换数据区中数据的变化能更好的
   理解 XVid I 帧编码处理过程。
// */

379     uint16_t x, y;

381     pEnc->mbParam.m_rounding_type = 1;
```



```
382     pEnc->current->rounding_type = pEnc->mbParam.m_rounding_type;
383     pEnc->current->coding_type = I_VOP;
/*      第 381 行和第 382 行设定 rounding_type, XVid 设定 I 帧的 rounding_type 为 1, P 帧的
        rounding_type 交替为 0 和 1, 这样半像素插值和运动补偿有更好的效果, 相对于 MPEG1 和 MPEG2
        恒为 0 的情况, 图像质量有比较大的提高。
// */
385     pEnc->current->y_scale = get_dc_scaler(pEnc->current->quant, 1);
386     pEnc->current->uv_scale = get_dc_scaler(pEnc->current->quant, 0);
387     pEnc->current->mb_width = pEnc->mbParam.mb_width;
/*      第 385 行到第 386 行计算几个变量, 原始的 XVid 码率控制做到宏块级, 因此每一个宏块的 DC Scale
        不同, 所以要在宏块级计算, 现在我们是在帧级做码率控制, 所以可以把 DC Scale 提前, 避
        免在每一个宏块中计算 DC Scale, 节省很多时空资源。
        第 387 行 mb_width 是为了参数传递方便多加到里面去的, 在此一并赋初值。
//      */

389     BitstreamWriteVolHeader(bs, &pEnc->mbParam, pEnc->current); // 写 VOL 头

391     BitstreamPad(bs); // 码流字节对齐。

393     BitstreamWriteVopHeader(bs, &pEnc->mbParam, pEnc->current, pEnc->current->quant);

395     for (y = 0; y < mb_height; y++)
396     {
397         for (x = 0; x < mb_width; x++)
398         {
399             MACROBLOCK *pMB = &pEnc->current->mbs[x + y * pEnc->mbParam.mb_width];

401             CodeIntraMB(pEnc, pMB);

403             MBTransQuantIntra(&pEnc->mbParam, pEnc->current, pMB, x, y, dct_codes, qcoeff);

405 //             MBPrediction(pEnc->current, x, y, pEnc->mbParam.mb_width, qcoeff);
406             predict_acdc2(pMB, pEnc->current, x, y, qcoeff);
407             pMB->cbp = calc_cbp(qcoeff);

409             CodeBlockIntra(pEnc->current, pMB, qcoeff, bs);
410         }
411     }
/*      第 399 行取得当前宏块缓冲区的首地址, 需要在此缓冲区中保存很多计算变量。
        第 401 行复位宏块缓冲区中的变量值。
```

第 403 行做前向和重构计算，包括拷贝到 dct_codes 缓冲区，DCT 变换，量化。

第 406 行做 DC 系数预测

第 407 行计算 cbp 值

第 409 行实质性编码码流。

I 帧编码整个流程很清晰，步骤也比较少，函数简单明了。

```
// */
```

```
413    __asm emms
```

```
/*    第 413 行因为用到了 MMX 指令，在这最后的地方做一次复位，不用在中间每一个用 MMX 指令的地方  
      都调用这条指令复位，节省时空资源。
```

```
// */
```

```
415    BitstreamPadAlways(bs); // 码流字节对齐。
```

```
417    pEnc->current->length = (BitstreamPos(bs) - bits) / 8; // 返回帧长度
```

```
419    pEnc->fMvPrevSigma = -1;
```

```
420    pEnc->mbParam.m_fcode = 2; // 初始赋值为 2，定义矢量搜索范围不超过 32 整像素。
```

```
422    return 1;
```

```
423 }
```

```
425 static __inline void updateFcode(Statistics * sStat, Encoder * pEnc)
```

```
426 {
```

```
427     float fSigma;
```

```
428     int iSearchRange;
```

```
430     if (sStat->iMvCount == 0)
```

```
431         sStat->iMvCount = 1;
```

```
433     fSigma = (float) sqrt((float) sStat->iMvSum / sStat->iMvCount);
```

```
435     iSearchRange = 16 << pEnc->mbParam.m_fcode;
```

```
437     if ((3.0 * fSigma > iSearchRange) && (pEnc->mbParam.m_fcode <= 5) )
```

```
438         pEnc->mbParam.m_fcode++;
```

```
440     else if ((5.0 * fSigma < iSearchRange)
```

```
441         && (4.0 * pEnc->fMvPrevSigma < iSearchRange)
```

```
442         && (pEnc->mbParam.m_fcode >= 2) )
```

```
443         pEnc->mbParam.m_fcode--;
```

```
445     pEnc->fMvPrevSigma = fSigma;
446 }
/* 第 425 行到第 446 行用当前帧的运动矢量个数和运动矢量的平方和来计算下一帧的搜索范围，即用当
   前帧的运动情况来指导下一帧的搜索范围，自适应调节功能。
// */

448 static int FrameCodeP(Encoder * pEnc, Bitstream * bs) // P 帧编码总控函数
449 {
450     int bits = BitstreamPos(bs);

452     DECLARE_ALIGNED_MATRIX(dct_codes, 6, 64, int16_t, CACHE_LINE);
453     DECLARE_ALIGNED_MATRIX(qcoeff, 6, 64, int16_t, CACHE_LINE);
/* 第 352 行到第 453 行声明了 P 帧编码过程中的核心交换数据区，以此数据区为中心，调用相关的
   过程函数按照 MP4 标准分步骤来处理，多跟踪几次这两个核心交换数据区中数据的变化能更好
   的理解 XVid P 帧编码处理过程。
// */

455     int x, y;
456     FRAMEINFO *const current = pEnc->current;
457     FRAMEINFO *const reference = pEnc->reference;
458     MBParam * const pParam = &pEnc->mbParam;
459     int mb_width = pParam->mb_width;
460     int mb_height = pParam->mb_height;

462     IMAGE *pRef = &reference->image;
/* 第 456 行到第 462 行定义一些中间变量，改变了一些引用形式，为的是程序可读性更好，代码更简
   洁，容易读懂理解，性能也有一定的提高。
// */

464     image_setedges(pRef, pParam->edged_width, pParam->edged_height,
465                   pParam->width, pParam->height); // 扩展图像边界

467     pParam->m_rounding_type = 1 - pParam->m_rounding_type;
468     current->rounding_type = pParam->m_rounding_type;
469     current->fcode = pParam->m_fcode;

471     image_interpolate(pRef, &pEnc->f_refh, &pEnc->f_refv, &pEnc->f_refhv, //半像素插值
472                     pParam->edged_width, pParam->edged_height, current->rounding_type);
/* 第 464 行到第 472 行代码其实是做准备工作。
```

```
// */
474     current->sStat.iMvSum = current->sStat.iMvCount = 0;

476     current->coding_type = P_VOP;

478     pEnc->current->y_scale = get_dc_scaler(pEnc->current->quant, 1);
479     pEnc->current->uv_scale = get_dc_scaler(pEnc->current->quant, 0);
480     pEnc->current->mb_width = pEnc->mbParam.mb_width;

482     MotionEstimation(&pEnc->mbParam, current, reference, // 运动估计, 超耗时的部分
483                     &pEnc->f_refh, &pEnc->f_refv, &pEnc->f_refhv);

485     BitstreamWriteVopHeader(bs, &pEnc->mbParam, current, current->quant);

487     for (y = 0; y < mb_height; y++)
488     {
489         for (x = 0; x < mb_width; x++)
490         {
491             MACROBLOCK *pMB = &current->mbs[x + y * pParam->mb_width];

493             if(pMB->mode == MODE_INTER)
494             {
495                 MBMotionCompensation(pMB, x, y, &reference->image, &pEnc->f_refh,
496                                     &pEnc->f_refv, &pEnc->f_refhv, &current->image, dct_codes,
497                                     pParam->edged_width, current->rounding_type);

499                 pMB->cbp = MBTransQuantInter(&pEnc->mbParam, current, pMB, x, y,
500                                             dct_codes, qcoeff);

502                 if(pMB->cbp) // 正常的 P 帧编码
503                 {
504                     BitstreamPutBit0(bs);
505                     CodeBlockInter(current, pMB, qcoeff, bs);
506                     continue;
507                 }
508                 else if(pMB->mvs.x | pMB->mvs.y) // 简化的 P 帧编码
509                 {
510                     BitstreamPutBit0(bs);
511                     CodeBlockInters(current, pMB, bs);
```

```
513             continue;
514         }
515         else //if (pMB->mode == MODE_NOT_CODED) //P 帧中的不编码宏块
516         {
517             BitstreamPutBit1(bs);
518             continue;
519         }
520     }
521     else if (pMB->mode == MODE_INTRA) // P 帧中的 I 块, 多了第 530 行代码。
522     {
523         CodeIntraMB(pEnc, pMB);
524         MBTransQuantIntra(&pEnc->mbParam, current, pMB, x, y, dct_codes, qcoeff);

526 //         MBPrediction(pEnc->current, x, y, pEnc->mbParam.mb_width, qcoeff);
527         predict_acdc2(pMB, current, x, y, qcoeff);
528         pMB->cbp = calc_cbp(qcoeff);

530         BitstreamPutBit0(bs);
531         CodeBlockIntra(current, pMB, qcoeff, bs);

533         continue;
534     }
535     else //if (pMB->mode == MODE_NOT_CODED)
536     {
537         int temp = pParam->edged_width;

539         transfer16x16_copy(current->image.y + 16 * (x + y * temp),
540             reference->image.y + 16 * (x + y * temp), temp);

542         temp /= 2;

544         transfer8x8_copy(current->image.u + 8 * (x + y * temp),
545             reference->image.u + 8 * (x + y * temp), temp);

547         transfer8x8_copy(current->image.v + 8 * (x + y * temp),
548             reference->image.v + 8 * (x + y * temp), temp);

550         BitstreamPutBit1(bs);
551     }
```

/* 注意第 535 行到第 550 行表示不编码宏块, 和第 515 行到第 519 行不编码宏块好像有些不同, 其感恩的心, 感谢生命中的每一个人。

实是相同的，第 495 行 MBMotionCompensation() 函数做的工作和第 537 行到第 548 行代码做的工作一样，行成预测帧。

```
// */
552     }
553 }

555     __asm emms

557     updateFcode(&current->sStat, pEnc);

559     BitstreamPadAlways(bs);

561     current->length = (BitstreamPos(bs) - bits) / 8;

563     return 1;
564 }
```

第三章 帧级处理程序

3.1 文件列表

类型	名称	大小
	image.h	640 bytes
	image.c	11261 bytes
	interpolate8x8.h	8298 bytes

3.2 image.h 文件

3.2.1 功能描述

帧级预处理函数原型的声明，包括帧内存申请和释放，帧数据输入，帧边缘扩展，帧半像素插值，帧指针交换等。

3.2.2 文件注释

```
1  #ifndef _IMAGE_H_
2  #define _IMAGE_H_

4  int32_t image_create(IMAGE * image, uint32_t edged_width, uint32_t edged_height);

6  void image_destroy(IMAGE * image, uint32_t edged_width);

8  void image_swap(IMAGE * image1, IMAGE * image2);

10 void image_setedges(IMAGE * image, uint32_t edged_width, uint32_t edged_height,
11                     uint32_t width, uint32_t height);

13 void image_interpolate(const IMAGE* refn, IMAGE* refh, IMAGE* refv, IMAGE* refhv,
14                       uint32_t edged_width, uint32_t edged_height, uint32_t rounding);

16 int image_input(IMAGE * image, uint32_t width, int height, uint32_t edged_width,
17                uint8_t * src, int src_stride);

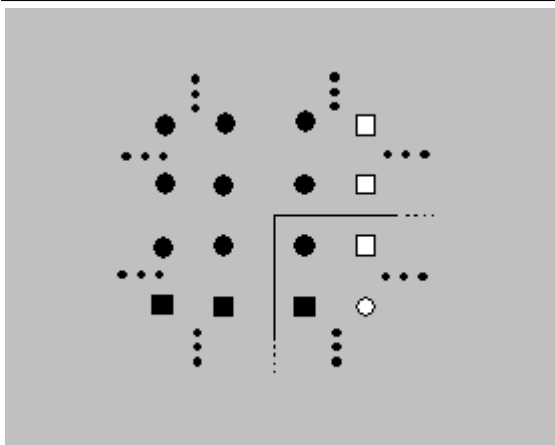
19 #endif
```

3.3 image.c 文件

3.3.1 功能描述

帧级预处理函数的实现，包括帧内存申请和释放，帧 YV12 数据输入，帧边缘扩展，帧半像素插值，帧指针交换等。

```
/*
```

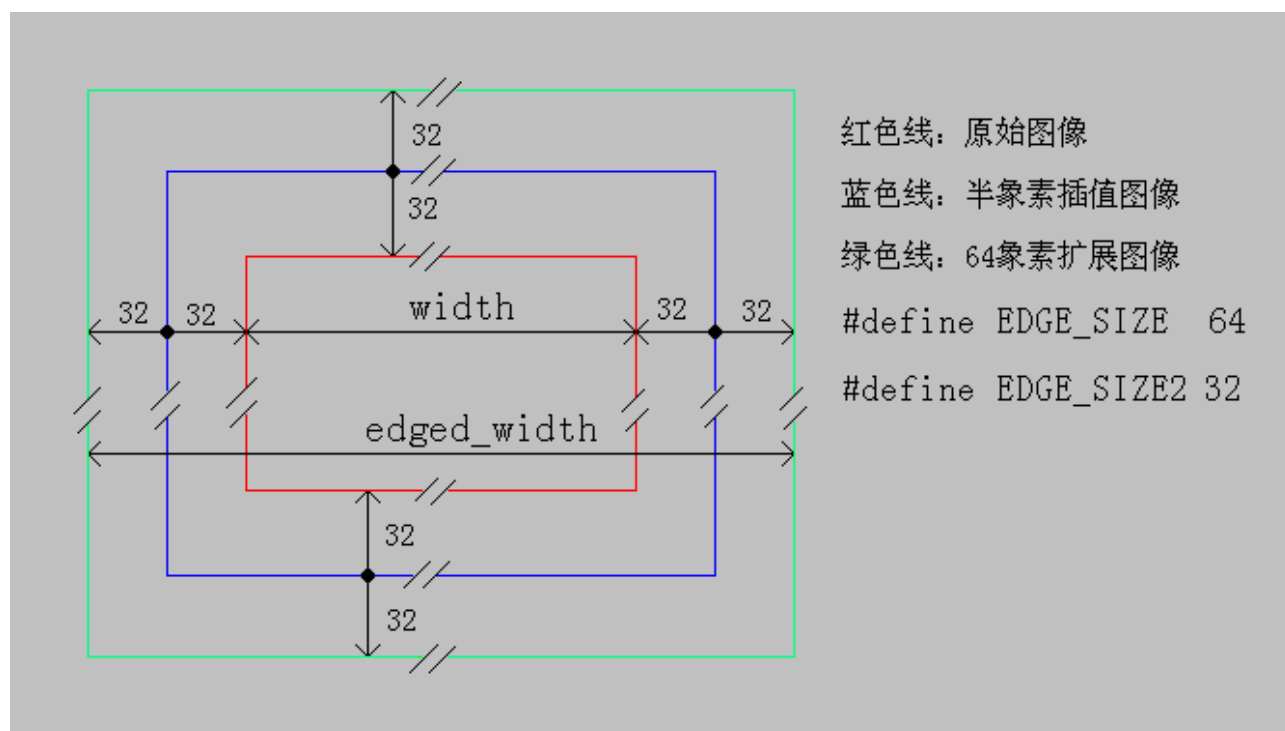


图七 边界扩展像素关系示意图

这是左上角，左边和上边 边界扩展时各像素值的关系，同一种图案表示同一个像素值。目前 XVid 边界扩展 64 像素，实际使用 32 像素来支持无限制运动矢量。

```
// */
```

```
/*
```

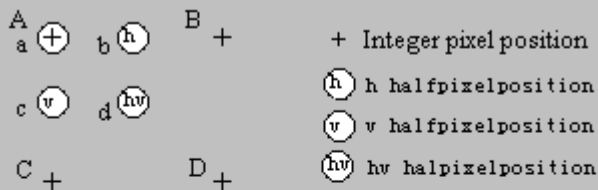


图八 帧内存分配示意图

帧内存分配示意图简单示意原始帧缓存，32 半像素插值帧缓存，64 像素边界扩展帧缓存的相对位置关系，计算图像座标时要清楚明白它们之间的关系，不能计算错相对偏移量。

```
// */
```

```
/*
```

a=A

b=(A+B+1-rounding_control)/2

c=(A+C+1-rounding_control)/2

d=(A+B+C+D+2-rounding_control)/2

图九 半像素插值位置关系示意图

```
// */
```

```
/*
```

做运动估计时，当前宏块要比较整像素点位置，还要比较半像素点位置，按照半像素点位置特征分为三个半像素平面，每行水平半像素位置点放到 h 半平面，每列垂直半像素位置点放到 v 半平面，每行列半像素交叉位置点放到 hv 平面。这样独立存放整像素和半像素位置点后，预测像素平面和编码像素平面有一样的长宽，一样的偏移，一样的 stride，便于统一处理。

因为 XVid 上下左右方向各扩展边界 64 像素，半像素插值只插值计算 32 像素，因此上下左右边界点不会越界。

```
// */
```

3.3.2 文件注释

```
1  #include <stdlib.h>
2  #include <string.h>

4  #include "../portab.h"
5  #include "../global.h"

7  #include "../utils/mem_align.h"

9  #include "image.h"
10 #include "interpolate8x8.h"

12 #define SAFETY 64 // 为了安全起见，冗余的字节数目

14 int32_t image_create(IMAGE * image, uint32_t edged_width, uint32_t edged_height)
15 {
16     const uint32_t edged_width2 = edged_width / 2;
17     const uint32_t edged_height2 = edged_height / 2;

19     image->y = xvid_malloc(edged_width * (edged_height + 1) + SAFETY, CACHE_LINE);
20     memset(image->y, 0, edged_width * (edged_height + 1) + SAFETY);
```

```
22     image->u = xvid_malloc(edged_width2 * edged_height2 + SAFETY, CACHE_LINE);
23     memset(image->u, 0, edged_width2 * edged_height2 + SAFETY);

25     image->v = xvid_malloc(edged_width2 * edged_height2 + SAFETY, CACHE_LINE);
26     memset(image->v, 0, edged_width2 * edged_height2 + SAFETY);

28     image->y += EDGE_SIZE * edged_width + EDGE_SIZE;
29     image->u += EDGE_SIZE2 * edged_width2 + EDGE_SIZE2;
30     image->v += EDGE_SIZE2 * edged_width2 + EDGE_SIZE2;

32     return 0;
33 }
/* 第 14 行到第 33 行分配图像的 YUV 缓冲区，注意是上下左右各方向都扩展了 64 像素，再把缓冲区清
   零，然后在第 28 行到第 30 行把 YUV 分量指针指到原始图像起始像素点位置。
// */

35 void image_destroy(IMAGE * image, uint32_t edged_width)
36 {
37     const uint32_t edged_width2 = edged_width / 2;

39     if (image->y)
40     {
41         xvid_free(image->y - (EDGE_SIZE * edged_width + EDGE_SIZE));
42         image->y = NULL;
43     }
44     if (image->u)
45     {
46         xvid_free(image->u - (EDGE_SIZE2 * edged_width2 + EDGE_SIZE2));
47         image->u = NULL;
48     }
49     if (image->v)
50     {
51         xvid_free(image->v - (EDGE_SIZE2 * edged_width2 + EDGE_SIZE2));
52         image->v = NULL;
53     }
54 }
/* 第 35 行到第 54 行释放图像 YUV 分量动态分配的内存，注意申请内存时返回的指针有偏移，释放时要
   先补偿偏移，再释放动态分配的内存。
// */

56 void image_swap(IMAGE * image1, IMAGE * image2)
57 {
58     SWAP(uint8_t*, image1->y, image2->y);
59     SWAP(uint8_t*, image1->u, image2->u);
60     SWAP(uint8_t*, image1->v, image2->v);
61 }
```

/* 第 56 行到第 61 行交换两个图像 YUV 分量的指针，避免 copy 数据需要的巨量操作。

// */

```
63 int image_input(IMAGE* image, uint32_t width, int height, uint32_t edged_width,
64                 uint8_t * src, int src_stride)
65 {
66     int y;

68     const int width2 = width/2;
69     const int height2 = height/2;

71     const int edged_width2 = edged_width/2;
72     const int src_stride2=src_stride/2;

74     uint8_t *y_dst=image->y;
75     uint8_t *u_dst=image->u;
76     uint8_t *v_dst=image->v;

78     uint8_t *y_src=src;
79     uint8_t *v_src=src + src_stride*height;
80     uint8_t *u_src=src + src_stride*height + src_stride2*height2;

82     for (y = height; y; y--) // 对 Y 分量间隔为 edged_width
83     {
84         memcpy(y_dst, y_src, width);
85         y_src += src_stride;
86         y_dst += edged_width;
87     }

89     for (y = height2; y; y--) // 对 U 分量间隔为 edged_width2
90     {
91         memcpy(u_dst, u_src, width2);
92         u_src += src_stride2;
93         u_dst += edged_width2;
94     }

96     for (y = height2; y; y--) // 对 V 分量间隔为 edged_width2
97     {
98         memcpy(v_dst, v_src, width2);
99         v_src += src_stride2;
100        v_dst += edged_width2;
101    }

103    return 0;
104 }
```

/* 第 63 行到第 104 行处理输入图像，把从 YUV 文件中读出来的连续 YUV 图像数据分段间隔存放在 YUV

缓冲区中。对 Y 分量相临行数据间隔 128byte, 对 UV 分量相临行间隔 64byte。

// */

```
106 void image_setedges(IMAGE * image, uint32_t edged_width, uint32_t edged_height,
107                     uint32_t width, uint32_t height)
108 {
109     const uint32_t edged_width2 = edged_width / 2;
110     uint32_t width2;
111     uint32_t i;
112     uint8_t *dst;
113     uint8_t *src;

115     dst = image->y - (EDGE_SIZE + EDGE_SIZE * edged_width);
116     src = image->y;

118     width2 = width/2;

120     for (i = 0; i < EDGE_SIZE; i++)    // 向上扩展图像 Y 分量
121     {
122         memset(dst, *src, EDGE_SIZE); // 整个左上角填图像起始点像素值。
123         memcpy(dst + EDGE_SIZE, src, width); // 正上方填上边界线对应像素值
124         memset(dst + edged_width - EDGE_SIZE, *(src + width - 1), EDGE_SIZE);
125                                         // 整个右上角填图像上边界线最右边像素值
126         dst += edged_width;
127     }

128     for (i = 0; i < height; i++)    // 向左右方向扩展 Y 分量
129     {
130         memset(dst, *src, EDGE_SIZE); // 左边填每对应行像素起始点值
131         memset(dst + edged_width - EDGE_SIZE, src[width - 1], EDGE_SIZE);
132                                         // 右边填每对应行像素结束点值
133         dst += edged_width;
134         src += edged_width;
135     }

136     src -= edged_width;
137     for (i = 0; i < EDGE_SIZE; i++)    // 向下扩展图像 Y 分量
138     {
139         memset(dst, *src, EDGE_SIZE); // 左小角填图像下边界线起始点像素值
140         memcpy(dst + EDGE_SIZE, src, width); // 正下方填图像下边界线对应点像素值
141         memset(dst + edged_width - EDGE_SIZE, *(src + width - 1), EDGE_SIZE);
142                                         // 右下方填图像下边界线结束点像素值
143         dst += edged_width;
144     }

145     /* U */ // U 分量和 Y 分量等同办法填充扩展边界, 但要注意扩展范围减半
```

```
146     dst = image->u - (EDGE_SIZE2 + EDGE_SIZE2 * edged_width2);
147     src = image->u;

149     for (i = 0; i < EDGE_SIZE2; i++)
150     {
151         memset(dst, *src, EDGE_SIZE2);
152         memcpy(dst + EDGE_SIZE2, src, width2);
153         memset(dst + edged_width2 - EDGE_SIZE2, *(src + width2 - 1), EDGE_SIZE2);
154         dst += edged_width2;
155     }

157     for (i = 0; i < height / 2; i++)
158     {
159         memset(dst, *src, EDGE_SIZE2);
160         memset(dst + edged_width2 - EDGE_SIZE2, src[width2 - 1], EDGE_SIZE2);
161         dst += edged_width2;
162         src += edged_width2;
163     }
164     src -= edged_width2;
165     for (i = 0; i < EDGE_SIZE2; i++)
166     {
167         memset(dst, *src, EDGE_SIZE2);
168         memcpy(dst + EDGE_SIZE2, src, width2);
169         memset(dst + edged_width2 - EDGE_SIZE2, *(src + width2 - 1), EDGE_SIZE2);
170         dst += edged_width2;
171     }

173     /* V */ // U 分量和 Y 分量等同办法填充扩展边界, 但要注意扩展范围减半
174     dst = image->v - (EDGE_SIZE2 + EDGE_SIZE2 * edged_width2);
175     src = image->v;

177     for (i = 0; i < EDGE_SIZE2; i++)
178     {
179         memset(dst, *src, EDGE_SIZE2);
180         memcpy(dst + EDGE_SIZE2, src, width2);
181         memset(dst + edged_width2 - EDGE_SIZE2, *(src + width2 - 1), EDGE_SIZE2);
182         dst += edged_width2;
183     }

185     for (i = 0; i < height / 2; i++)
186     {
187         memset(dst, *src, EDGE_SIZE2);
188         memset(dst + edged_width2 - EDGE_SIZE2, src[width2 - 1], EDGE_SIZE2);
189         dst += edged_width2;
190         src += edged_width2;
191     }
```

```
192     src -= edged_width2;
193     for (i = 0; i < EDGE_SIZE2; i++)
194     {
195         memset(dst, *src, EDGE_SIZE2);
196         memcpy(dst + EDGE_SIZE2, src, width2);
197         memset(dst + edged_width2 - EDGE_SIZE2, *(src + width2 - 1), EDGE_SIZE2);
198         dst += edged_width2;
199     }
200 }

202 #ifndef INTERPOLATE_ACC

204 void image_interpolate(const IMAGE * refn, IMAGE * refh, IMAGE * refv, IMAGE * refhv,
205                       uint32_t edged_width, uint32_t edged_height, uint32_t rounding)
206 {
207     const uint32_t offset = EDGE_SIZE2 * (edged_width + 1);
208     /* 第 207 行 offset 计算由图像起始点位置到向外扩展 32 像素以字节为单位的偏移量，因为图像上
        下左右方向已扩展，所以宽度方向是 edged_width。EDGE_SIZE2*edged_width 是往图像正上方
        偏移 EDGE_SIZE2 行，+1 表示是向左偏移 EDGE_SIZE2 像素(字节)
        // */
209     const uint32_t stride_add = 7 * edged_width;
210     /* 程序以 8x8 小方块为单位做插值，每次递增量为 8*edged_width，因为每次插值是从左到右已经
        递增了 edged_width-EDGE_SIZE2*2 个像素，所以下面程序中增量用两步修改。(h_ptr +=
        EDGE_SIZE; h_ptr += stride_add;)
        // */
211     uint8_t *n_ptr, *h_ptr, *v_ptr, *hv_ptr;
212     uint32_t x, y;

213     n_ptr = refn->y;
214     h_ptr = refh->y;
215     v_ptr = refv->y;
216     hv_ptr = refhv->y;

217     n_ptr -= offset;
218     h_ptr -= offset;
219     v_ptr -= offset;
220     hv_ptr -= offset;
221     /* 第 213 行到第 221 行把指针指向图像起始点向外扩展 32 像素处。
        // */
222     for (y = 0; y < (edged_height - EDGE_SIZE); y += 8)
223     {
224         for (x = 0; x < (edged_width - EDGE_SIZE); x += 8)
225         {
226             interpolate8x8_halfpel_h(h_ptr, n_ptr, edged_width, rounding);
227             interpolate8x8_halfpel_v(v_ptr, n_ptr, edged_width, rounding);
228             interpolate8x8_halfpel_hv(hv_ptr, n_ptr, edged_width, rounding);
229         }
230     }
231 }
```

```
/*          第 227 行到第 229 行直接调用 8x8 小块插值函数计算插值。
// */
231         n_ptr += 8;
232         h_ptr += 8;
233         v_ptr += 8;
234         hv_ptr += 8;
235     }

237     h_ptr += EDGE_SIZE;
238     v_ptr += EDGE_SIZE;
239     hv_ptr += EDGE_SIZE;
240     n_ptr += EDGE_SIZE;

242     h_ptr += stride_add;
243     v_ptr += stride_add;
244     hv_ptr += stride_add;
245     n_ptr += stride_add;
/*          第 237 行到第 245 行修改指针，垂直方向增量到下一个 8x8 块。
// */

246     }
247 }
249 #else
250 /*  计算公式:
251      $(i+j)/2 = (i+j+1)/2 - (i^j)\&1$ 
252      $(i+j+k+1+2)/4 = (s+t+1)/2 - (ij|kl)\&st$ 
253      $(i+j+k+1+1)/4 = (s+t+1)/2 - (ij\&kl)|st$ 
254     with  $s=(i+j+1)/2$ ,  $t=(k+l+1)/2$ ,  $ij = i^j$ ,  $kl = k^l$ ,  $st = s^t$ .
255 // */
256 __declspec(align(16)) static unsigned char mmx_one_16[16] = {
257     1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

259 void image_interpolate(const IMAGE * refn, IMAGE * refh, IMAGE * refv, IMAGE * refhv,
260     uint32_t edged_width, uint32_t edged_height, uint32_t rounding)
261 {
/*      汇编版的半像素插值每次插值计算两行，每次把 h, v, hv 三分量同时计算出来。
      程序完全遵守上面的计算公司，但开发完之后为避免指令相关又调整了一下指令先后顺序，因此
      很多地方逻辑上看起来不连续，很混乱的样子，小心仔细阅读才行。
      部分不大熟悉的汇编指令请参考附录中的解释。
// */
262     int xitem = EDGE_SIZE2 * (edged_width + 1);
263     int yitem = (edged_height - EDGE_SIZE) / 2;

265     uint8_t *prefn=refn->y-xitem;
266     uint8_t *prefh=refh->y-xitem;
267     uint8_t *prefv=refv->y-xitem;
```

```
268     uint8_t *prefhv=refhv->y-xitem;

270     if(rounding==0)
271     {
272         __asm
273         {
274             pushad

276             mov eax, prefn
277             mov ebx, prefh
278             mov ecx, prefv
279             mov edx, prefhv

281             movdqa xmm7, [mmx_one_16]

283             mov edi, edged_width
284             sal edi, 1      // edi = edged_width * 2,
285 loop2:
286             mov esi, edi
287             sar esi, 5
288             mov xitem, esi // xitem = edged_width / 16
289             sub xitem, 4
290             mov esi, edi
291             sar esi, 1      // esi = edged_width, stride
292 loop1:
293             movdqa xmm0, [eax]
294             movdqu xmm1, [eax+1]
295             movdqa xmm2, [eax+esi]
296             movdqu xmm3, [eax+esi+1]

298             movdqa xmm4, xmm0
299             movdqa xmm5, xmm2

301             pavgb  xmm0, xmm1 // xmm0 = s = (i+j+1)/2
302             pavgb  xmm2, xmm3 // xmm2 = t = (k+l+1)/2

304             pxor   xmm1, xmm4 // xmm1 = ij = i^j, xmm4 = [eax]
305             pxor   xmm3, xmm5 // xmm3 = kl = k^l, xmm5 = [eax+esi]

307             movdqa [ebx], xmm0 //---interpolate(h1)  // xmm6 = [eax]
308
309             pavgb  xmm4, xmm5

311             movdqa [ebx+esi], xmm2  //---interpolate(h2)

313             movdqa xmm6, xmm0 // xmm4 = xmm0 = s
```



```
315          movdqa [ecx], xmm4 //---interpolate(v1)

317          pxor   xmm6, xmm2 // xmm4 = st = s^t

319          por    xmm1, xmm3 // (ij|kl)

321          pavgb  xmm0, xmm2 // xmm0 = (s+t+1)/2

323          pand   xmm1, xmm6 // (ij|kl)&st
324          pand   xmm1, xmm7 // 取低位
325          psubb  xmm0, xmm1 // (s+t+1)/2 - (ij|kl)&st

327          movdqa [edx], xmm0 //---interpolate(hv1) // 第一行数据处理完

329          movdqa xmm0, [eax+edi]
330          movdqu xmm1, [eax+edi+1]
331          movdqa xmm4, xmm0
332          pavgb  xmm0, xmm1 // xmm0 = s = (i+j+1)/2
333          pxor   xmm1, xmm4 // xmm1 = ij = i^j, xmm6 = [eax+edi]

335          movdqa xmm6, xmm0
336          pavgb  xmm4, xmm5
337          pxor   xmm6, xmm2 // xmm4 = st = s^j
338          movdqa [ecx+esi], xmm4 //---interpolate(v2)

340          por    xmm3, xmm1
341          pavgb  xmm2, xmm0 // xmm2 = (s+t+1)/2
342          pand   xmm3, xmm6
343          pand   xmm3, xmm7
344          psubb  xmm2, xmm3
345          movdqa [edx+esi], xmm2 //---interpolate(hv2)

347          add eax, 16
348          add ebx, 16
349          add ecx, 16
350          add edx, 16

352          dec xitem

354          jne loop1

356          add esi, 64
357          add eax, esi
358          add ebx, esi
359          add ecx, esi
```

```
360          add edx, esi

362          dec yitem

364          jne loop2

366          popad
367      }
368  }
369  else // if(rounding==1)
370  {
371      __asm
372      {
373          pushad

375          mov eax, prefn
376          mov ebx, prefh
377          mov ecx, prefv
378          mov edx, prefhv

380          mov edi, edged_width
381          sal edi, 1 // edi = edged_width * 2
382 loop4:
383          mov esi, edi
384          sar esi, 5
385          mov xitem, esi // xitem = edged_width / 16
386          sub xitem, 4
387          mov esi, edi
388          sar esi, 1 // esi = edged_width, stride
389 loop3:
390          movdqa xmm0, [eax] // i
391          movdqu xmm1, [eax+1] // j
392          movdqa xmm2, [eax+esi] // k
393          movdqu xmm3, [eax+esi+1] // l

395          movdqa xmm4, xmm0
396          movdqa xmm5, xmm2

398          pavgb xmm4, xmm1 // xmm4 = s = (i+j+1)/2
399          pavgb xmm5, xmm3 // xmm5 = t = (k+l+1)/2

401          movdqa xmm6, xmm4
402          movdqa xmm7, xmm4
403          pavgb xmm6, xmm5 // xmm6 = (s+t+1)/2
404          pxor xmm7, xmm5 // xmm7 = st
```

```

406      pxor    xmm1, xmm0 // xmm1 = i j = i^j
407      pxor    xmm3, xmm2 // xmm3 = k l = k^l
408      pand    xmm1, [mmx_one_16] // 取低位
409      pand    xmm3, [mmx_one_16] // 取低位
410      psubb   xmm4, xmm1 // (s+t+1)/2 - (i^j)&1
411      psubb   xmm5, xmm3 // (s+t+1)/2 - (i^j)&1

413      pand    xmm1, xmm3      // (i+j+k+l+1)/4 = (s+t+1)/2 - (ij&kl) | st
414      movdqa  [ebx], xmm4     //---interpolate(h1)
415      por     xmm1, xmm7
416      movdqa  [ebx+esi], xmm5 //---interpolate(h2)
417      pand    xmm1, [mmx_one_16]
418      movdqa  xmm4, xmm0
419      psubb   xmm6, xmm1
420      pavgb   xmm0, xmm2
421      movdqa  [edx], xmm6     // interpolate(hv1)

423      pxor    xmm4, xmm2
424      movdqa  xmm1, [eax+edi]
425      pand    xmm4, [mmx_one_16]
426      psubb   xmm0, xmm4
427      movdqa  [ecx], xmm0    // interpolate(v1) //xmm2=[eax+esi], xmm3=k1,

429      movdqa  xmm4, xmm2
430      movdqa  xmm0, xmm2     //xmm0=xmm2=[eax+esi], xmm1=[eax+edi], xmm3=k1
431      pxor    xmm4, xmm1
432      pavgb   xmm0, xmm1
433      pand    xmm4, [mmx_one_16]
434      psubb   xmm0, xmm4
435      movdqa  [ecx+esi], xmm0 // interpolate(v2) //

437      movdqu  xmm4, [eax+esi+1]
438      movdqu  xmm0, [eax+edi+1] //

440      movdqa  xmm5, xmm0
441      pavgb   xmm2, xmm4
442      pavgb   xmm0, xmm1
443      movdqa  xmm4, xmm2

445      pxor    xmm5, xmm1     // xmm0=s, xmm2=t, xmm5=ij, xmm3=k1
446      pavgb   xmm2, xmm0
447      pxor    xmm4, xmm0     // xmm2=(s+t+1)/2, xmm4 = st

449      pand    xmm5, xmm3
450      por     xmm5, xmm4
451      pand    xmm5, [mmx_one_16]

```

```
452         psubb xmm2, xmm5
453         movdqa [edx+esi], xmm2 // hv2

455         add eax, 16
456         add ebx, 16
457         add ecx, 16
458         add edx, 16

460         dec xitem

462         jne loop3

464         add esi, 64
465         add eax, esi
466         add ebx, esi
467         add ecx, esi
468         add edx, esi

470         dec yitem

472         jne loop4

474         popad
475     }
476 }
477 }

471 #endif
```

3.4 interpolate8x8.h 文件

3.4.1 功能描述

XVid 编码器宏块级半像素插值(h, v, hv)三函数的实现, 包括 C 语言和 PC 汇编语言版本。因为 XVid 帧级半像素插值直接调用宏块级半像素插值相应函数, 所以归类到帧级处理大类中。

3.4.2 文件注释

```
1  #ifndef _INTERPOLATE8X8_H_
2  #define _INTERPOLATE8X8_H_

4  #ifdef INTERPOLATE_ACC

6  __declspec(aligned(16)) static unsigned char mmx_one_8[8] = {1, 1, 1, 1, 1, 1, 1, 1};

8  #define COPY_H_SSE_RND0          \ //直接使用 pavgb 指令, 两行并行处理(rounding=0)
```

```

9      __asm movq mm0, [eax] \
10     __asm movq mm1, [eax+edx] \
11     __asm pavgb mm0, [eax+1] \
12     __asm pavgb mm1, [eax+edx+1] \
13     __asm movq [ecx], mm0 \
14     __asm movq [ecx+edx], mm1

16 #define COPY_H_SSE_RND1 \ // 使用计算公式 (i+j)/2 = ( i+j+1 )/2 - (i^j)&1
17     __asm movq mm0, [eax] \ // 当前行进 mm0
18     __asm movq mm1, [eax+edx] \ // 下一行进 mm1
19     __asm movq mm4, mm0 \ // 备份 mm0 到 mm4
20     __asm movq mm5, mm1 \ // 备份 mm1 到 mm5
21     __asm movq mm2, [eax+1] \ // 当前行偏移一个字节进 mm2
22     __asm movq mm3, [eax+edx+1] \ // 下一行偏移一个字节进 mm3
23     __asm pavgb mm0, mm2 \ // 当前行水平方向求和平均(i+j+1)/2 进 mm0
24     __asm pavgb mm1, mm3 \ // 下一行水平方向求和平均(i+j+1)/2 进 mm1
25     __asm pxor mm2, mm4 \ // 当前行水平方向(i^j)进 mm2
26     __asm pxor mm3, mm5 \ // 下一行水平方向(i^j)进 mm3
27     __asm pand mm2, mm7 \ // 当前行 (i^j) &1
28     __asm pand mm3, mm7 \ // 下一行 (i^j) &1
29     __asm psubb mm0, mm2 \ // 当前行(i+j)/2
30     __asm psubb mm1, mm3 \ // 下一行(i+j)/2
31     __asm movq [ecx], mm0 \ // 传送到目的地址
32     __asm movq [ecx+edx], mm1 // 传送到目的地址

34 static __inline void interpolate8x8_halfpel_h(uint8_t* const dst,
35         const uint8_t* const src, const uint32_t stride, const uint32_t rounding)
36 {
37     __asm mov eax, rounding
38     __asm mov ecx, dst
39     __asm test eax, eax // 判断 rounding 值是否为 0
40     __asm mov eax, src
41     __asm mov edx, stride

43     __asm jnz rounding1

45     COPY_H_SSE_RND0

47     __asm lea eax, [eax+2*edx] // 更新 eax = eax + 2*edx, 即 eax 偏移两行
48     __asm lea ecx, [ecx+2*edx] // 更新 ecx = ecx + 2*edx, 即 ecx 偏移两行
/* 从逻辑上来讲, 应该把 第 47 行和第 48 行放到 COPY_H_SSE_RND0 宏定义中去的, 但是最后一次不
   用这两条语句, 为节省时间和 code size 就直接写在程序中。
// */

49     COPY_H_SSE_RND0

```

```
51     __asm lea eax, [eax+2*edx]
52     __asm lea ecx, [ecx+2*edx]
53     COPY_H_SSE_RND0

55     __asm lea eax, [eax+2*edx]
56     __asm lea ecx, [ecx+2*edx]
57     COPY_H_SSE_RND0

59     return ;

61 rounding1:

63     __asm movq mm7, [mmx_one_8]
64     COPY_H_SSE_RND1

66     __asm lea eax, [eax+2*edx]
67     __asm lea ecx, [ecx+2*edx]
68     COPY_H_SSE_RND1

70     __asm lea eax, [eax+2*edx]
71     __asm lea ecx, [ecx+2*edx]
72     COPY_H_SSE_RND1

74     __asm lea eax, [eax+2*edx]
75     __asm lea ecx, [ecx+2*edx]
76     COPY_H_SSE_RND1

78     return;
79 }

81 #define COPY_V_SSE_RND0          \\\ 处理流程和 COPY_H_SSE_RND0 相同
82     __asm movq mm0, [eax]        \
83     __asm movq mm1, [eax+edx]    \
84     __asm pavgb mm0, mm1         \
85     __asm pavgb mm1, [eax+2*edx] \
86     __asm movq [ecx], mm0        \
87     __asm movq [ecx+edx], mm1

89 #define COPY_V_SSE_RND1          \\\ 处理流程和 COPY_H_SSE_RND1 相同
90     __asm movq mm0, mm2          \
91     __asm movq mm1, [eax]        \
92     __asm movq mm2, [eax+edx]    \
93     __asm movq mm4, mm0          \
94     __asm movq mm5, mm1          \
95     __asm pavgb mm0, mm1         \
96     __asm pxor mm4, mm1          \
```

```
97     __asm pavgb mm1, mm2      \  
98     __asm pxor mm5, mm2      \  
99     __asm pand mm4, mm7      \  
100    __asm pand mm5, mm7      \  
101    __asm psubb mm0, mm4      \  
102    __asm movq [ecx], mm0     \  
103    __asm psubb mm1, mm5      \  
104    __asm movq [ecx+edx], mm1  
  
106 static __inline void interpolate8x8_halfpel_v(uint8_t* const dst,  
107         const uint8_t* const src, const uint32_t stride, const uint32_t rounding)  
108 {  
109     __asm mov eax, rounding  
110     __asm mov ecx, dst  
111     __asm test eax, eax  
112     __asm mov eax, src  
113     __asm mov edx, stride  
  
115     __asm jnz rounding1  
  
117     COPY_V_SSE_RND0  
  
119     __asm lea eax, [eax+2*edx]  
120     __asm lea ecx, [ecx+2*edx]  
121     COPY_V_SSE_RND0  
  
123     __asm lea eax, [eax+2*edx]  
124     __asm lea ecx, [ecx+2*edx]  
125     COPY_V_SSE_RND0  
  
127     __asm lea eax, [eax+2*edx]  
128     __asm lea ecx, [ecx+2*edx]  
129     COPY_V_SSE_RND0  
  
131     return;  
  
133 rounding1:  
134     __asm movq mm7, [mmx_one_8]  
135     __asm movq mm2, [eax]  
136     __asm add eax, edx  
  
138     COPY_V_SSE_RND1  
  
140     __asm lea eax, [eax+2*edx]  
141     __asm lea ecx, [ecx+2*edx]  
142     COPY_V_SSE_RND1
```

```

144     __asm lea eax, [eax+2*edx]
145     __asm lea ecx, [ecx+2*edx]
146     COPY_V_SSE_RND1

148     __asm lea eax, [eax+2*edx]
149     __asm lea ecx, [ecx+2*edx]
150     COPY_V_SSE_RND1

152     return;
153 }

```

/* 计算公式

$$(i+j+k+1+3)/4 = (s+t+1)/2 - (ij\&k1)\&st$$

$$(i+j+k+1+2)/4 = (s+t+1)/2 - (ij|k1)\&st$$

$$(i+j+k+1+1)/4 = (s+t+1)/2 - (ij\&k1)|st$$

$$(i+j+k+1+0)/4 = (s+t+1)/2 - (ij|k1)|st$$

with $s=(i+j+1)/2$, $t=(k+1+1)/2$, $ij = i^{\wedge}j$, $k1 = k^{\wedge}1$, $st = s^{\wedge}t$.

// */

```

155 #define COPY_HV_SSE_RND0 \
156     __asm lea eax, [eax+edx] \
157     \
158     __asm movq mm0, [eax] \ // 不严格意义 mm0=k;
159     __asm movq mm1, [eax+1] \ // 不严格意义 mm1=1;
160     \
161     __asm movq mm6, mm0 \ // 不严格意义 mm6=k;
162     __asm pavgb mm0, mm1 \ // 不严格意义 mm0=t=(k+1+1)/2;
163     __asm lea eax, [eax+edx] \
164     __asm pxor mm1, mm6 \ // 不严格意义 mm1=k^1;
165     \ // 不严格意义 mm2=s=(i+j+1)/2; mm3=i^j;
166     __asm por mm3, mm1 \ // 不严格意义 mm3=(ij|kj)
167     __asm movq mm6, mm2 \ // 不严格意义 mm6=mm2
168     __asm pxor mm6, mm0 \ // 不严格意义 mm6 = st
169     __asm pand mm3, mm6 \ // 不严格意义 mm3 = (ij|kj)&st
170     __asm pavgb mm2, mm0 \ // 不严格意义 mm2 = (s+t+1)/2
171     __asm pand mm3, mm7 \ // 不严格意义 mm3 = mm3&0x0101010101010101
172     __asm psubb mm2, mm3 \ // 不严格意义 mm2 = (s+t+1)/2 - (ij|k1)&st
173     \
174     __asm movq [ecx], mm2 \
175     \
176     __asm movq mm2, [eax] \
177     __asm movq mm3, [eax+1] \
178     __asm movq mm6, mm2 \
179     __asm pavgb mm2, mm3 \
180     __asm lea ecx, [ecx+edx] \
181     __asm pxor mm3, mm6 \

```



```
182                                     \
183     __asm por mm1, mm3              \
184     __asm movq mm6, mm0             \
185     __asm pxor mm6, mm2             \
186     __asm pand mm1, mm6            \
187     __asm pavgb mm0, mm2           \
188                                     \
189     __asm pand mm1, mm7            \
190     __asm psubb mm0, mm1           \
191                                     \
192     __asm movq [ecx], mm0

194 #define COPY_HV_SSE_RND1          \
195     __asm lea eax, [eax+edx]        \
196                                     \
197     __asm movq mm0, [eax]           \
198     __asm movq mm1, [eax+1]         \
199                                     \
200     __asm movq mm6, mm0             \
201     __asm pavgb mm0, mm1            \
202     __asm lea eax, [eax+edx]        \
203     __asm pxor mm1, mm6            \
204                                     \
205     __asm pand mm3, mm1             \
206     __asm movq mm6, mm2            \
207     __asm pxor mm6, mm0            \
208     __asm por mm3, mm6              \
209     __asm pavgb mm2, mm0            \
210     __asm pand mm3, mm7            \
211     __asm psubb mm2, mm3            \
212                                     \
213     __asm movq [ecx], mm2           \
214                                     \
215     __asm movq mm2, [eax]           \
216     __asm movq mm3, [eax+1]         \
217     __asm movq mm6, mm2            \
218     __asm pavgb mm2, mm3            \
219     __asm lea ecx, [ecx+edx]        \
220     __asm pxor mm3, mm6            \
221                                     \
222     __asm pand mm1, mm3            \
223     __asm movq mm6, mm0            \
224     __asm pxor mm6, mm2            \
225     __asm por mm1, mm6             \
226     __asm pavgb mm0, mm2           \
227     __asm pand mm1, mm7            \
```

```
228     __asm psubb mm0, mm1      \
229                               \
230     __asm movq [ecx], mm0

232 static __inline void interpolate8x8_halfpel_hv(uint8_t* const dst,
233         const uint8_t* const src, const uint32_t stride, const uint32_t rounding)
234 {
235     __asm mov eax, rounding
236     __asm mov ecx, dst
237     __asm test eax, eax
238     __asm mov eax, src
239     __asm mov edx, stride

241     __asm movq mm7, [mmx_one_8]

243     __asm movq mm2, [eax]
244     __asm movq mm3, [eax+1]
245     __asm movq mm6, mm2
246     __asm pavgb mm2, mm3
247     __asm pxor mm3, mm6      ; mm2/mm3 ready // 不严格意义 mm2=(i+j+1)/2; mm3=i^j;

249     __asm jnz rounding1

251     COPY_HV_SSE_RND0

253     __asm add ecx, edx
254     COPY_HV_SSE_RND0

256     __asm add ecx, edx
257     COPY_HV_SSE_RND0

259     __asm add ecx, edx
260     COPY_HV_SSE_RND0

262     return ;

264 rounding1:
265     COPY_HV_SSE_RND1

267     __asm add ecx, edx
268     COPY_HV_SSE_RND1

270     __asm add ecx, edx
271     COPY_HV_SSE_RND1

273     __asm add ecx, edx
```

```
274     COPY_HV_SSE_RND1

276     return;
277 }

280 #else

/* 增加程序局部变量 r=1-rounding 或 r=2-rounding 是为了去除判断分支操作中中断 CPU 指令流水线。
// */
/* interpolater8x8_halfpel_h()函数水平方向半像素插值，简单的数学计算。
// */
283 static __inline void interpolate8x8_halfpel_h(uint8_t* const dst,
284         const uint8_t* const src, const uint32_t stride, const uint32_t rounding)
285 {
286     uint32_t j;

288     int r = 1 - rounding;

290     for (j = 0; j < 8*stride; j+=stride)
291     {
292         dst[j + 0] = (uint8_t)((src[j + 0] + src[j + 1] + r)>>1);
293         dst[j + 1] = (uint8_t)((src[j + 1] + src[j + 2] + r)>>1);
294         dst[j + 2] = (uint8_t)((src[j + 2] + src[j + 3] + r)>>1);
295         dst[j + 3] = (uint8_t)((src[j + 3] + src[j + 4] + r)>>1);
296         dst[j + 4] = (uint8_t)((src[j + 4] + src[j + 5] + r)>>1);
297         dst[j + 5] = (uint8_t)((src[j + 5] + src[j + 6] + r)>>1);
298         dst[j + 6] = (uint8_t)((src[j + 6] + src[j + 7] + r)>>1);
299         dst[j + 7] = (uint8_t)((src[j + 7] + src[j + 8] + r)>>1);
300     }
301 }

/* interpolater8x8_halfpel_v()函数垂直方向半像素插值，简单的数学计算。
// */
303 static __inline void interpolate8x8_halfpel_v(uint8_t* const dst,
304         const uint8_t* const src, const uint32_t stride, const uint32_t rounding)
305 {
306     uint32_t j;
307
308     int r = 1 - rounding;
309
310     for (j = 0; j < 8*stride; j+=stride)
311     {
312         dst[j + 0] = (uint8_t)((src[j + 0] + src[j + stride + 0] + r)>>1);
313         dst[j + 1] = (uint8_t)((src[j + 1] + src[j + stride + 1] + r)>>1);
314         dst[j + 2] = (uint8_t)((src[j + 2] + src[j + stride + 2] + r)>>1);
```

```
315         dst[j + 3] = (uint8_t)((src[j + 3] + src[j + stride + 3] + r)>>1);
316         dst[j + 4] = (uint8_t)((src[j + 4] + src[j + stride + 4] + r)>>1);
317         dst[j + 5] = (uint8_t)((src[j + 5] + src[j + stride + 5] + r)>>1);
318         dst[j + 6] = (uint8_t)((src[j + 6] + src[j + stride + 6] + r)>>1);
319         dst[j + 7] = (uint8_t)((src[j + 7] + src[j + stride + 7] + r)>>1);
320     }
321 }

/* interpolater8x8_halfpel_hv()函数 2x2 中心点半像素插值，简单的数学计算。
// */
323 static __inline void interpolate8x8_halfpel_hv(uint8_t* const dst,
324         const uint8_t* const src, const uint32_t stride, const uint32_t rounding)
325 {
326     uint32_t j;
327
328     int r = 2- rounding;
329
330     for (j = 0; j < 8*stride; j+=stride)
331     {
332         dst[j+0]=(uint8_t)((src[j+0]+src[j+1]+src[j+stride+0]+src[j+stride+1]+r)>>2);
333         dst[j+1]=(uint8_t)((src[j+1]+src[j+2]+src[j+stride+1]+src[j+stride+2]+r)>>2);
334         dst[j+2]=(uint8_t)((src[j+2]+src[j+3]+src[j+stride+2]+src[j+stride+3]+r)>>2);
335         dst[j+3]=(uint8_t)((src[j+3]+src[j+4]+src[j+stride+3]+src[j+stride+4]+r)>>2);
336         dst[j+4]=(uint8_t)((src[j+4]+src[j+5]+src[j+stride+4]+src[j+stride+5]+r)>>2);
337         dst[j+5]=(uint8_t)((src[j+5]+src[j+6]+src[j+stride+5]+src[j+stride+6]+r)>>2);
338         dst[j+6]=(uint8_t)((src[j+6]+src[j+7]+src[j+stride+6]+src[j+stride+7]+r)>>2);
339         dst[j+7]=(uint8_t)((src[j+7]+src[j+8]+src[j+stride+7]+src[j+stride+8]+r)>>2);
340     }
341 }

343 #endif

345 #endif
```

第四章 运动估计

4.1 文件列表

类型	名称	大小
	motion.h	573 bytes
	sad.h	6649 bytes
	motion_comp.c	3178 bytes
	estimation_pvop.c	18048 bytes

4.2 motion.h 文件

4.2.1 功能描述

运动估计和运动补偿函数原型的声明。

4.2.2 文件注释

```
1  #ifndef _MOTION_H_
2  #define _MOTION_H_

4  extern const uint32_t roundtab_79[4];

6  void MotionEstimation(MBParam* const pParam, FRAMEINFO* const current,
7                      FRAMEINFO* const reference, const IMAGE* const pRefH,
8                      const IMAGE* const pRefV, const IMAGE* const pRefHV);

10 void MBMotionCompensation(MACROBLOCK* const mb, const uint32_t i, const uint32_t j,
11                          const IMAGE* const ref, const IMAGE* const refh, const IMAGE* const refv,
12                          const IMAGE* const refhv, IMAGE * const cur, int16_t * dct_codes,
13                          const uint32_t edged_width, const int32_t rounding);

15 #endif
```

4.3 sad.h 文件

4.3.1 功能描述

16x16 宏块和 8x8 小块 SAD 值计算, 16x16 均值绝对误差和计算, 包括 C 语言和 PC 汇编语言版本, 带 s 后缀的汇编宏定义表示是简单版(small), 最后一个调用, 少几行汇编代码。

4.3.2 文件注释

```
1  #ifndef _ENCODER_SAD_H_
2  #define _ENCODER_SAD_H_
```

```
4  #ifdef SAD_ACC

6  #define SAD_8x8_SSE          \
7      __asm movq mm0, [eax]    \
8      __asm movq mm1, [eax+ecx] \
9      __asm psadbw mm0, [edx]   \ // 核心计算指令
10     __asm psadbw mm1, [edx+ecx] \ // 核心计算指令
11     __asm add eax, ebx         \
12     __asm add edx, ebx         \
13     __asm paddusw mm5, mm0     \
14     __asm paddusw mm6, mm1

16 static __inline uint32_t sad8(const uint8_t* const cur, const uint8_t* const ref,
17                               const uint32_t stride) // 计算 8x8 小块的 SAD 值, 利用 psadbw 指令直接计算
18 {
19     int sad;

21     __asm mov ecx, stride ; Stride
22     __asm mov eax, cur    ; Src1
23     __asm mov edx, ref    ; Src2
24     __asm lea ebx, [ecx+ecx]

26     __asm pxor mm5, mm5 ; accum1
27     __asm pxor mm6, mm6 ; accum2

29     SAD_8x8_SSE
30     SAD_8x8_SSE
31     SAD_8x8_SSE

33     __asm movq mm0, [eax]
34     __asm movq mm1, [eax+ecx]
35     __asm psadbw mm0, [edx]
36     __asm psadbw mm1, [edx+ecx]

38     __asm paddusw mm5, mm0
39     __asm paddusw mm6, mm1

41     __asm paddusw mm6, mm5

43     __asm movd sad, mm6 // movd sad, mm6

45     return sad;
46 }

48 #define SAD_16x16_SSE2          \
49     __asm movdqu xmm0, [edx]    \
```

```
50     __asm movdqu xmm1, [edx+ecx] \
51     __asm lea edx, [edx+2*ecx] \
52     __asm movdqa xmm2, [eax] \
53     __asm movdqa xmm3, [eax+ecx] \
54     __asm lea eax, [eax+2*ecx] \
55     __asm psadbw xmm0, xmm2 \
56     __asm psadbw xmm1, xmm3 \
57     __asm paddusw xmm4, xmm0 \
58     __asm paddusw xmm5, xmm1

60 #define SAD_16x16_SSE2s \
61     __asm movdqu xmm0, [edx] \
62     __asm movdqu xmm1, [edx+ecx] \
63     __asm movdqa xmm2, [eax] \
64     __asm movdqa xmm3, [eax+ecx] \
65     __asm psadbw xmm0, xmm2 \
66     __asm psadbw xmm1, xmm3 \
67     __asm paddusw xmm4, xmm0 \
68     __asm paddusw xmm5, xmm1

70 static __inline uint32_t sad16(const uint8_t* const cur, const uint8_t* const ref,
71                               const uint32_t stride)
72 {
73     int sad;

75     __asm mov eax, cur ; cur (assumed aligned)
76     __asm mov edx, ref ; ref
77     __asm mov ecx, stride ; stride

79     __asm pxor xmm4, xmm4 ; accum
80     __asm pxor xmm5, xmm5 ; accum

82     SAD_16x16_SSE2
83     SAD_16x16_SSE2
84     SAD_16x16_SSE2
85     SAD_16x16_SSE2
86     SAD_16x16_SSE2
87     SAD_16x16_SSE2
88     SAD_16x16_SSE2
89     SAD_16x16_SSE2s

91     __asm paddusw xmm4, xmm5
92     __asm pshufd xmm7, xmm4, 00000010b
93     __asm paddusw xmm4, xmm7
94     __asm pextrw eax, xmm4, 0
```

```

96     __asm    mov sad, eax

98     return sad;
99 }

/* 第 70 行到第 99 行计算 16x16 大块的 SAD 值, 利用 psadbw 指令直接计算。
   注意 psadbw 使用 128bit 位运算时, 高低位计算的和放在 bit[79..64]和 bit[15..0]中, 所以相对于
   sad8() 多了第 92 行和第 93 行
// */

101 #define MEAN_16x16_SSE2          \
102     __asm    movdqu xmm0, [eax]    \
103     __asm    movdqu xmm1, [eax+ecx] \
104     __asm    lea  eax, [eax+2*ecx]  \
105     __asm    psadbw xmm0, xmm7     \
106     __asm    psadbw xmm1, xmm7     \
107     __asm    paddusw xmm4, xmm0     \
108     __asm    paddusw xmm5, xmm1

110 #define MEAN_16x16_SSE2s          \
111     __asm    movdqu xmm0, [eax]    \
112     __asm    movdqu xmm1, [eax+ecx] \
113     __asm    psadbw xmm0, xmm7     \
114     __asm    psadbw xmm1, xmm7     \
115     __asm    paddusw xmm4, xmm0     \
116     __asm    paddusw xmm5, xmm1

118 static __inline uint32_t dev16(const uint8_t * const cur, const uint32_t stride)
119 {
120     int sad;

122     __asm    mov  eax, cur    ; src
123     __asm    mov  ecx, stride ; stride

125     __asm    pxor xmm4, xmm4  ; accum
126     __asm    pxor xmm5, xmm5  ; accum
127     __asm    pxor xmm7, xmm7  ; zero

129     MEAN_16x16_SSE2
130     MEAN_16x16_SSE2
131     MEAN_16x16_SSE2
132     MEAN_16x16_SSE2

134     MEAN_16x16_SSE2
135     MEAN_16x16_SSE2
136     MEAN_16x16_SSE2
137     MEAN_16x16_SSE2s

```



```

139     __asm    paddusw  xmm4, xmm5

141     __asm    mov  eax, cur          ; src again

143     __asm    pshufd   xmm7, xmm4, 10b
144     __asm    pxor     xmm5, xmm5      ; zero accum
145     __asm    paddusw  xmm7, xmm4
146     __asm    pxor     xmm4, xmm4      ; zero accum
147     __asm    psrlw    xmm7, 8         ; => Mean
148     __asm    pshufdw  xmm7, xmm7, 0   ; replicate Mean
149     __asm    packuswb xmm7, xmm7
150     __asm    pshufd   xmm7, xmm7, 00000000b

152     MEAN_16x16_SSE2
153     MEAN_16x16_SSE2
154     MEAN_16x16_SSE2
155     MEAN_16x16_SSE2

157     MEAN_16x16_SSE2
158     MEAN_16x16_SSE2
159     MEAN_16x16_SSE2
160     MEAN_16x16_SSE2s

162     __asm    paddusw  xmm4, xmm5

164     __asm    pshufd   xmm7, xmm4, 10b
165     __asm    paddusw  xmm7, xmm4
166     __asm    pextrw   eax, xmm7, 0

168     __asm    mov  sad, eax

170     return sad;
171 }
/* 第 118 行到第 171 行计算一阶均值距离, 就是先计算 16x16 大块的均值, 再计算各个像素点相对于这
   个均值的一阶距离。相对于 sad16() 函数, 多了计算 16x16 大块的均值步骤。
   /**/

173 static __inline uint32_t sad16v(const uint8_t* const cur, const uint8_t* const ref,
174                                const uint32_t stride, int32_t *sad)
175 {
176     sad[0] = sad8(cur, ref, stride);
177     sad[1] = sad8(cur + 8, ref + 8, stride);
178     sad[2] = sad8(cur + 8*stride, ref + 8*stride, stride);
179     sad[3] = sad8(cur + 8*stride + 8, ref + 8*stride + 8, stride);

```

```
181     return sad[0]+sad[1]+sad[2]+sad[3];
182 }

185 #else

188 static __inline uint32_t sad16(const uint8_t* const cur, const uint8_t* const ref,
189                               const uint32_t stride) // 计算 16x16 大块的 SAD 值
190 {
191     uint32_t sad = 0;
192     uint32_t j;
193     uint8_t const *ptr_cur = cur;
194     uint8_t const *ptr_ref = ref;

196     for (j = 0; j < 16; j++)
197     {
198         sad += abs(ptr_cur[0] - ptr_ref[0]);
199         sad += abs(ptr_cur[1] - ptr_ref[1]);
200         sad += abs(ptr_cur[2] - ptr_ref[2]);
201         sad += abs(ptr_cur[3] - ptr_ref[3]);
202         sad += abs(ptr_cur[4] - ptr_ref[4]);
203         sad += abs(ptr_cur[5] - ptr_ref[5]);
204         sad += abs(ptr_cur[6] - ptr_ref[6]);
205         sad += abs(ptr_cur[7] - ptr_ref[7]);
206         sad += abs(ptr_cur[8] - ptr_ref[8]);
207         sad += abs(ptr_cur[9] - ptr_ref[9]);
208         sad += abs(ptr_cur[10] - ptr_ref[10]);
209         sad += abs(ptr_cur[11] - ptr_ref[11]);
210         sad += abs(ptr_cur[12] - ptr_ref[12]);
211         sad += abs(ptr_cur[13] - ptr_ref[13]);
212         sad += abs(ptr_cur[14] - ptr_ref[14]);
213         sad += abs(ptr_cur[15] - ptr_ref[15]);

215 //         if (sad >= best_sad)
216 //             return sad;

218         ptr_cur += stride;
219         ptr_ref += stride;
220     }
221     return sad;
222 }

224 static __inline uint32_t sad8(const uint8_t* const cur, const uint8_t* const ref,
225                               const uint32_t stride) // 计算 8x8 小块的 SAD 值
226 {
```

```
227     uint32_t sad = 0;
228     uint32_t j;
229     uint8_t const *ptr_cur = cur;
230     uint8_t const *ptr_ref = ref;

232     for (j = 0; j < 8; j++)
233     {
234         sad += abs(ptr_cur[0] - ptr_ref[0]);
235         sad += abs(ptr_cur[1] - ptr_ref[1]);
236         sad += abs(ptr_cur[2] - ptr_ref[2]);
237         sad += abs(ptr_cur[3] - ptr_ref[3]);
238         sad += abs(ptr_cur[4] - ptr_ref[4]);
239         sad += abs(ptr_cur[5] - ptr_ref[5]);
240         sad += abs(ptr_cur[6] - ptr_ref[6]);
241         sad += abs(ptr_cur[7] - ptr_ref[7]);

243         ptr_cur += stride;
244         ptr_ref += stride;
245     }
246     return sad;
247 }

249 static __inline uint32_t dev16(const uint8_t * const cur, const uint32_t stride)
250 { // 计算 16x16 大块的一阶均值距离
251     uint32_t mean = 0;
252     uint32_t dev = 0;
253     uint32_t i, j;
254     uint8_t const *ptr_cur = cur;

256     for (j = 0; j < 16; j++)
257     {
258         for (i = 0; i < 16; i++)
259             mean += *(ptr_cur + i);
260         ptr_cur += stride;
261     }

263     mean /= (16 * 16);
264     ptr_cur = cur;

266     for (j = 0; j < 16; j++)
267     {
268         for (i = 0; i < 16; i++)
269             dev += abs(*(ptr_cur + i) - (int32_t) mean);

271         ptr_cur += stride;
272     }
```

```
274     return dev;
275 }

277 static __inline uint32_t sad16v(const uint8_t* const cur, const uint8_t* const ref,
278                                const uint32_t stride, int32_t *sad)
279 {
280     sad[0] = sad8(cur, ref, stride);
281     sad[1] = sad8(cur + 8, ref + 8, stride);
282     sad[2] = sad8(cur + 8*stride, ref + 8*stride, stride);
283     sad[3] = sad8(cur + 8*stride + 8, ref + 8*stride + 8, stride);

285     return sad[0]+sad[1]+sad[2]+sad[3];
286 }

288 #endif

290 #endif
```

4.4 motion_comp.c 文件

4.4.1 功能描述

运动补偿函数及其辅助函数的实现。

4.4.2 文件注释

```
1  #include "../portab.h"
2  #include "../global.h"
3  #include "../encoder.h"

5  #include "../utils/mem_transfer.h"
6  #include "../image/interpolate8x8.h"

8  #include "motion.h"

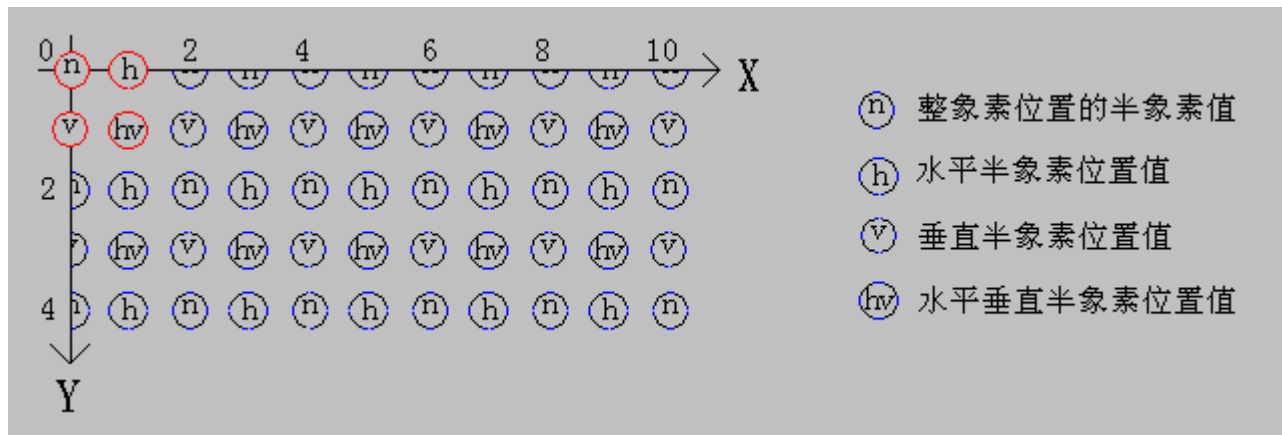
10 static __inline const uint8_t *get_ref(const uint8_t * const refn,
11                                         const uint8_t* const refh, const uint8_t * const refv,
12                                         const uint8_t* const refhv, const uint32_t x, const uint32_t y,
13                                         const int32_t dx, const int32_t dy, const int32_t stride)
14 {
15     switch (((dx & 1) << 1) + (dy & 1))
16     {
17     case 0:
18         return refn+(int)(((int)x+dx/2) + ((int)y + dy / 2) * (int)stride);
19     case 1:
```

```

20         return refv+(int)(((int)x+dx/2) + ((int)y + (dy - 1) / 2) * (int)stride);
21     case 2:
22         return refh+(int)(((int)x+(dx-1)/2) + ((int)y+ dy/2) * (int)stride);
23     default:
24         return refhv+(int)(((int)x+(dx-1)/2) + ((int)y+ (dy-1)/2) * (int)stride);
25     }
26 }

```

/* 第 10 行到第 26 行按照半像素运动矢量和整像素起始点坐标来决定参考帧的起始地址。



图十 n, h, v, hv 半像素位置整合平面坐标示意图

由图十可知, n 半像素位置 x 方向运动矢量为偶数, y 方向运动矢量为偶数;

h 半像素位置 x 方向运动矢量为奇数, y 方向运动矢量为偶数;

v 半像素位置 x 方向运动矢量为偶数, y 方向运动矢量为奇数;

hv 半像素位置 x 方向运动矢量为奇数, y 方向运动矢量为奇数;

所以第 15 行可以用运动矢量的奇偶性来区分 n, h, v, hv 半像素平面。

在图十中, 红色 n, h, v, hv 半像素坐标位置表示相应 n, h, v, hv 半像素独立平面坐标系原点(0, 0)

在 n, h, v, hv 整合平面中的相对位置, 由图可知 h, v, hv 三半像素坐标有 1 个半像素偏移, 所以在第 20 行第 22 行第 24 行中 dx, dy 有减 1 的修正计算。

从图十可知, 由 n, h, v, hv 整合平面分离出 n, h, v, hv 四个半像素坐标平面, 平面长宽减半, 所以第 18 行第 20 行第 22 行第 24 行 dx, dy 有除 2 的运算。

此函数实现了 n, h, v, hv 整合平面运动估计运动补偿使用的半像素计量单位到 n, h, v, hv 四个半像素独立平面运动估计运动补偿使用的像素计量单位的变换, 起到关键的桥接转换作用。

// */

```

28 static __inline void compensat16x16_interpolate(int16_t* const dct_codes,
29         uint8_t* const cur, const uint8_t* const ref, const uint8_t* const refh,
30         const uint8_t* const refv, const uint8_t* const refhv,
31         uint32_t x, uint32_t y, const int32_t dx, const int32_t dy,
32         const int32_t stride)
33 {
34     const uint8_t * ptr;
35
36     ptr = get_ref(ref, refh, refv, refhv, x, y, dx, dy, stride);

```

```
38     transfer_8tol6sub(dct_codes,      cur+y*stride+x,          ptr,          stride);
39     transfer_8tol6sub(dct_codes+64,  cur+y*stride+x+8,        ptr+8,          stride);
40     transfer_8tol6sub(dct_codes+128, cur+y*stride+x+8*stride, ptr+8*stride,  stride);
41     transfer_8tol6sub(dct_codes+192, cur+y*stride+x+8*stride+8, ptr+8*stride+8, stride);
42 }
/* 第 28 行到第 42 行把当前编码图像一个宏块的 Y 分量减预测图像适当宏块的 Y 分量的差保存到核心数
   据缓存区 dct_codes[] 中，同时把预测图像适当宏块的 Y 分量拷贝到当前图像适当位置中。每次处
   理一个 8x8 小块共处理四次。
// */

44 static __inline uint8_t *interpolate8x8_switch2(uint8_t* const buffer,
45          const uint8_t* const refn, const int x, const int y, const int dx,
46          const int dy, const uint32_t stride, const uint32_t rounding)
47 {
48     const uint8_t * const src = refn + (int)((y + (dy>>1)) * stride + x + (dx>>1));

50     switch (((dx & 1) << 1) + (dy & 1))
51     {
52     case 0:
53         return (uint8_t *)src;
54     case 1:
55         interpolate8x8_halfpel_v(buffer, src, stride, rounding);
56         break;
57     case 2:
58         interpolate8x8_halfpel_h(buffer, src, stride, rounding);
59         break;
60     default:
61         interpolate8x8_halfpel_hv(buffer, src, stride, rounding);
62         break;
63     }
64     return buffer;
65 }
/* 第 44 行到第 65 行色度分量半像素插值, 亮度分量已插值好分别存放在 refh, refv, refhv 中备用,
   色度分量即插值即使用不保存。
// */

67 void MBMotionCompensation(MACROBLOCK * const mb, const uint32_t i, const uint32_t j,
68          const IMAGE* const ref, const IMAGE* const refh, const IMAGE* const refv,
69          const IMAGE* const refhv, IMAGE* const cur, int16_t* dct_codes,
70          const uint32_t edged_width, const int32_t rounding)
71 {
72     uint8_t * const tmp = refv->u;
73     const int32_t stride = edged_width / 2;

75     int32_t dx = mb->mvs.x;
76     int32_t dy = mb->mvs.y;
```

```
78     compensatel6x16_interpolate(&dct_codes[0 * 64], cur->y, ref->y, refh->y, refv->y,
79         refhv->y, 16 * i, 16 * j, dx, dy, edged_width);

81     dx = (dx >> 1) + roundtab_79[dx & 0x3];
82     dy = (dy >> 1) + roundtab_79[dy & 0x3];

84     transfer_8tol6sub(&dct_codes[256], cur->u + 8 * j * stride + 8 * i,
85         interpolate8x8_switch2(tmp, ref->u, 8 * i, 8 * j, dx, dy, stride, rounding),
86         stride);

88     transfer_8tol6sub(&dct_codes[320], cur->v + 8 * j * stride + 8 * i,
89         interpolate8x8_switch2(tmp, ref->v, 8 * i, 8 * j, dx, dy, stride, rounding),
90         stride);
91 }
/* 第 67 行到第 91 行宏块运动补偿计算，第 78 行到第 79 行计算亮度分量运动补偿，第 84 行到第 90 行
   色度分量运动补偿。
   第 81 行和第 82 行中的 roundtab_79[] 数组是色度矢量修正值。
// */
```

4.5 estimation_pvop.c 文件

4.5.1 功能描述

运动估计函数及其辅助函数的实现。

4.5.2 文件注释

```
1  #include <math.h>
2  #include <string.h>

4  #include "../portab.h"
5  #include "../global.h"
6  #include "../encoder.h"

8  #include "../utils/mem_transfer.h"
9  #include "../image/interpolate8x8.h"

11 #include "motion.h"
12 #include "sad.h"

14 #define MV_MAX_ERROR    (4096 * 256)

16 #define MV16_INTER_BIAS 450
```

```
18 #define NEIGH_TEND_16X16      0.6
19 #define NEIGH_TEND_8X8        0.6
20 #define NEIGH_8X8_BIAS        40

22 #define INITIAL_SKIP_THRESH    6
23 #define FINAL_SKIP_THRESH      50
24 #define MAX_SAD00_FOR_SKIP     20
25 #define MAX_CHROMA_SAD_FOR_SKIP 22
/* 第 14 行到第 25 行定义一些控制变量常数宏。
// */

27 #define EVEN(A)      (((A)<0?(A)+1:(A)) & ~1)

29 #define MVEqual(A,B) ( ((A).x)==(B).x && ((A).y)==(B).y )

31 static const VECTOR zeroMV = { 0, 0 };

33 #define iDiamondSize 2

35 typedef struct
36 {
37     int max_dx, min_dx, max_dy, min_dy; /* maximum search range */

39     int32_t iMinSAD;          /* smallest SADs found so far */
40     VECTOR currentMV;         /* best vectors found so far */
41     int temp[4];              /* temporary space */
42     unsigned int dir;         /* 'direction', set when better vector is found */

44     uint32_t rounding;        /* rounding type in use */
45     VECTOR predMV;            /* vector which predicts current vector */
46     const uint8_t * RefP[6];   /* reference pictures - N, V, H, HV, cU, cV */
47     const uint8_t * Cur;       /* current picture */

49     uint8_t * RefQ;            /* temporary space for interpolations */
50     uint32_t lambda16;         /* how much vector bits weight */
51     uint32_t iEdgedWidth;      /* picture's stride */
52     uint32_t iFcode;          /* current fcode */

54 } SearchData;
```



```
56 static const int r_mvtab[64] = { // 运动矢量权重表
57     12, 12, 12, 12, 12, 12, 12, 12,
58     12, 12, 12, 12, 12, 12, 12, 12,
59     12, 12, 12, 12, 12, 12, 12, 12,
60     12, 12, 12, 12, 12, 12, 12, 12,
61     12, 11, 11, 11, 11, 11, 11, 10,
62     10, 10, 10, 10, 10, 10, 10, 10,
63     10, 10, 10, 10, 10, 9, 9, 9,
64     7, 7, 7, 6, 4, 3, 2, 1
65 };

67 const uint32_t roundtab_79[4] = { 0, 1, 0, 0 }; // 色度运动矢量修正数组

69 const int xvid_me_lambda_vec16[32] = // lamda 权重表
70 {
71     0, (int)(0.5 * NEIGH_TEND_16X16 + 0.5),
72     (int)(1.0*NEIGH_TEND_16X16 + 0.5), (int)(1.5*NEIGH_TEND_16X16 + 0.5),
73     (int)(2.0*NEIGH_TEND_16X16 + 0.5), (int)(2.5*NEIGH_TEND_16X16 + 0.5),
74     (int)(5.0*NEIGH_TEND_16X16 + 0.5), (int)(7.0*NEIGH_TEND_16X16 + 0.5),
75     (int)(8.0*NEIGH_TEND_16X16 + 0.5), (int)(9.0*NEIGH_TEND_16X16 + 0.5),
76     (int)(10.0*NEIGH_TEND_16X16 + 0.5), (int)(11.0*NEIGH_TEND_16X16 + 0.5),
77     (int)(12.0*NEIGH_TEND_16X16 + 0.5), (int)(13.0*NEIGH_TEND_16X16 + 0.5),
78     (int)(14.0*NEIGH_TEND_16X16 + 0.5), (int)(15.0*NEIGH_TEND_16X16 + 0.5),
79     (int)(16.0*NEIGH_TEND_16X16 + 0.5), (int)(17.0*NEIGH_TEND_16X16 + 0.5),
80     (int)(18.0*NEIGH_TEND_16X16 + 0.5), (int)(19.0*NEIGH_TEND_16X16 + 0.5),
81     (int)(20.0*NEIGH_TEND_16X16 + 0.5), (int)(21.0*NEIGH_TEND_16X16 + 0.5),
82     (int)(22.0*NEIGH_TEND_16X16 + 0.5), (int)(23.0*NEIGH_TEND_16X16 + 0.5),
83     (int)(24.0*NEIGH_TEND_16X16 + 0.5), (int)(25.0*NEIGH_TEND_16X16 + 0.5),
84     (int)(26.0*NEIGH_TEND_16X16 + 0.5), (int)(27.0*NEIGH_TEND_16X16 + 0.5),
85     (int)(28.0*NEIGH_TEND_16X16 + 0.5), (int)(29.0*NEIGH_TEND_16X16 + 0.5),
86     (int)(30.0*NEIGH_TEND_16X16 + 0.5), (int)(31.0*NEIGH_TEND_16X16 + 0.5)
87 };

88 static void __inline get_range(int32_t * const min_dx, int32_t * const max_dx,
89     int32_t * const min_dy, int32_t * const max_dy,
90     const uint32_t x, const uint32_t y,
91     const uint32_t width, const uint32_t height, const int fcode)
92 {
93     int k;
94     const int search_range = 1 << (4+fcode);
95     int high = search_range - 1;
```

```
96     int low = -search_range;

98     k = (int)(width - (x<<4))<<1;
99     *max_dx = MIN(high, k);
100    k = (int)(height - (y<<4))<<1;
101    *max_dy = MIN(high, k);

103    k = (-(int)((x+1)<<4))<<1;
104    *min_dx = MAX(low, k);
105    k = (-(int)((y+1)<<4))<<1;
106    *min_dy = MAX(low, k);
107 }

/* 第 88 行到第 107 行限定运动矢量的搜索范围不能超出图像半像素插值边界。
// */

109 static __inline uint32_t d_mv_bits(int x, int y, const VECTOR pred,
110                                   const uint32_t iFcode) // 计算运动矢量权重, 影响预测精度
111 {
112     unsigned int bits;

114     x -= pred.x;
115     bits = (x != 0 ? iFcode:0);
116     x = -abs(x);
117     x >>= (iFcode - 1);
118     bits += r_mvtab[x+63];

120     y -= pred.y;
121     bits += (y != 0 ? iFcode:0);
122     y = -abs(y);
123     y >>= (iFcode - 1);
124     bits += r_mvtab[y+63];

126     return bits;
127 }

129 static __inline const uint8_t *GetReference(const int x, const int y,
130                                             const SearchData * const data) // 根据运动矢量值返回预测块起始地址值
131 {
132     const int picture = ((x&1)<<1) | (y&1);
133     const int offset = (x>>1) + (y>>1)*data->iEdgedWidth;
```

```
134     return data->RefP[picture] + offset;
135 }

137 static __inline void ZeroMacroblockP(MACROBLOCK *pMB, const int32_t sad)//复位
138 {
139     pMB->mode = MODE_INTER;
140     pMB->mvs = zeroMV;
141     pMB->sad16 = pMB->sad8[0] = pMB->sad8[1] = pMB->sad8[2] = pMB->sad8[3] = sad;
142     pMB->cbp = 0;
143 }

145 static __inline int vector_repeats(const VECTOR * const pmv, const unsigned int i)
146 {
147     unsigned int j;
148     for (j = 0; j < i; j++)
149         if (MVequal(pmv[i], pmv[j])) return 1;
150     return 0;
151 }

/* 判断运动矢量是否和以前的相同, 如果相同那么以前已经计算过对应当前运动矢量的 SAD 值, 为节省
   时间这次就不用再计算。
// */

153 static __inline int make_mask(const VECTOR * const pmv, const unsigned int i,
154                               const unsigned int current)
155 {
156     unsigned int mask = 255, j;
157     for (j = 0; j < i; j++)
158     {
159         if (pmv[current].x == pmv[j].x)
160         {
161             if (pmv[current].y == pmv[j].y + iDiamondSize) mask &= ~4;
162             else if (pmv[current].y == pmv[j].y - iDiamondSize) mask &= ~8;
163         }
164         else if (pmv[current].y == pmv[j].y)
165         {
166             if (pmv[current].x == pmv[j].x + iDiamondSize) mask &= ~1;
167             else if (pmv[current].x == pmv[j].x - iDiamondSize) mask &= ~2;
168         }
169     }
170     return mask;
}
```

```
171 }
/* 返回整像素搜索时运动矢量的掩码，屏蔽那些已经计算过的运动矢量
// */
173 void CheckCandidateVector(const int x, const int y, SearchData * const data,
174         const unsigned int Direction)//计算当前运动矢量的 SAD 值，并比较记录相应结果
175 {
176     int32_t sad;
177     const uint8_t * Reference;
178     uint32_t t;

180     if ((x>data->max_dx) || (x<data->min_dx) || (y>data->max_dy) || (y<data->min_dy))
181         return;

183     Reference = GetReference(x, y, data);

185     t = d_mv_bits(x, y, data->predMV, data->iFcode);

187     sad = sad16(data->Cur, Reference, data->iEdgedWidth);
188     sad += (data->lambdal6 * t);

190     if (sad < data->iMinSAD)
191     {
192         data->dir = Direction;
193         data->iMinSAD = sad;
194         data->currentMV.x = x;
195         data->currentMV.y = y;
196     }
197 }

199 void CheckCandidate16no4v(const int x, const int y, SearchData* const data,
200         const unsigned int Direction)// 整像素搜索时计算 SAD 值，父函数限界运动矢量
201 {
202     const uint8_t * Reference = data->RefP[0] + (x>>1) + (y>>1)*data->iEdgedWidth;

204     uint32_t t = d_mv_bits(x, y, data->predMV, data->iFcode);

206     int32_t sad = sad16(data->Cur, Reference, data->iEdgedWidth);
207     sad += (data->lambdal6 * t);

209     if (sad < data->iMinSAD)
```

```
210     {
211         data->dir = Direction;
212         data->iMinSAD = sad;
213         data->currentMV.x = x;
214         data->currentMV.y = y;
215     }
216 }

218 void xvid_me_DiamondSearch(int x, int y, SearchData * const data, int bDirection)
219 {
220     int icount = -3;
221     /* 第 220 行限定整像素搜索最多 3 次。因为以我有限统计表明整像素搜索平均 2.5 轮就会退出，为
        避免浪费太多时间搜索需要 intra 编码的宏块，限定为 3 次。
        // */

222     unsigned int * const iDirection = &data->dir;

224     if ( (x > data->max_dx-8) || (x < data->min_dx+8)
225         || (y > data->max_dy-8) || (y < data->min_dy+8) )
226         return;
227     /* 第 224 行到第 226 行在限定最多搜索 3 次后，可以把运动矢量比较提前做大判断，减小中断 CPU 指
        令流水线的概率。
        // */

228     while (icount)
229     {
230         icount++;

232         *iDirection = 0;
233         if (bDirection & 1) CheckCandidate16no4v(x - iDiamondSize, y, data, 1);
234         if (bDirection & 2) CheckCandidate16no4v(x + iDiamondSize, y, data, 2);
235         if (bDirection & 4) CheckCandidate16no4v(x, y - iDiamondSize, data, 4);
236         if (bDirection & 8) CheckCandidate16no4v(x, y + iDiamondSize, data, 8);

238         if (*iDirection == 0)
239             break;
240
241         bDirection = *iDirection;
242         x = data->currentMV.x;
243         y = data->currentMV.y;
```

```
245         if (bDirection & 3) // our candidate is left or right
246         {
247             CheckCandidate16no4v(x, y - iDiamondSize, data, 4);
248             CheckCandidate16no4v(x, y + iDiamondSize, data, 8);
249         }
250         else // what remains here is up or down
251         {
252             CheckCandidate16no4v(x - iDiamondSize, y, data, 1);
253             CheckCandidate16no4v(x + iDiamondSize, y, data, 2);
254         }
255         bDirection |= *iDirection;
256         x = data->currentMV.x;
257         y = data->currentMV.y;
258     }
259 }

261 void CheckSubpelRefine(const int x, const int y, SearchData* const data,
262                       const unsigned int Direction) // 半像素搜索计算 SAD 值, 父函数限界运动矢量
263 {
264     const int picture = ((x&1)<<1) | (y&1);

266     const uint8_t* Reference = data->RefP[picture]+(x>>1)+(y>>1)*data->iEdgedWidth;

268     uint32_t t = d_mv_bits(x, y, data->predMV, data->iFcode);

270     int32_t sad = sad16(data->Cur, Reference, data->iEdgedWidth);
271     sad += (data->lambdal6 * t);

273     if (sad < data->iMinSAD)
274     {
275         data->dir = Direction;
276         data->iMinSAD = sad;
277         data->currentMV.x = x;
278         data->currentMV.y = y;
279     }
280 }

282 void CheckSubpelRefine_s(const int x, const int y, SearchData * const data)
// 最后一次半像素搜索计算 SAD 值不用记录方向信息。为节省时间单独简化成一个函数。
283 {
```

```
284     const int picture = ((x&1)<<1) | (y&1);

286     const uint8_t* Reference = data->RefP[picture]+(x>>1)+(y>>1)*data->iEdgedWidth;

288     uint32_t t = d_mv_bits(x, y, data->predMV, data->iFcode);

290     int32_t sad = sad16(data->Cur, Reference, data->iEdgedWidth);
291     sad += (data->lambdal6 * t);

293     if (sad < data->iMinSAD)
294     {
295         data->iMinSAD = sad;
296         data->currentMV.x = x;
297         data->currentMV.y = y;
298     }
299 }

301 void xvid_me_SubpelRefine(int x, int y, SearchData * const data)// 半像素搜索
302 {
303     if ( (x > data->max_dx-2) || (x < data->min_dx+2)
304         || (y > data->max_dy-2) || (y < data->min_dy+2) )
305         return;
/*    半像素搜索只有一轮，因此可以把运动矢量比较提前，减少中断 CPU 指令的概率
// */
307     data->dir =0;

309     CheckSubpelRefine(x, y - 1, data, 4);
310     CheckSubpelRefine(x, y + 1, data, 8);

312     CheckSubpelRefine(x - 1, y, data, 1);
313     CheckSubpelRefine(x + 1, y, data, 2);
/*    第 309 行到第 313 行先搜索 XY 坐标轴上的四个点
// */
315     switch(data->dir)
316     {
317     case 1:
318         CheckSubpelRefine_s(x - 1, y - 1, data);
319         CheckSubpelRefine_s(x - 1, y + 1, data);
320         break;
321     case 2:
```

```
322     CheckSubpelRefine_s(x + 1, y - 1, data);
323     CheckSubpelRefine_s(x + 1, y + 1, data);
324     break;
325 case 4:
326     CheckSubpelRefine_s(x - 1, y - 1, data);
327     CheckSubpelRefine_s(x + 1, y - 1, data);
328     break;
329 case 8:
330     CheckSubpelRefine_s(x - 1, y + 1, data);
331     CheckSubpelRefine_s(x + 1, y + 1, data);
332     break;
333 default:
334     break;
335 }
```

/* 第 317 行到第 332 行, 以最小 SAD 值对应的 XY 坐标轴点为基准, 搜索最临近的两个像素。根据最优点旁边是次优点的局部性原理, 可以少搜索比较几个点, 节省时间。

```
// */
```

```
336 }
```

```
338 unsigned int getMinFcode(const int MVmax) // 根据运动矢量来计算运动矢量的模
339 {
340     unsigned int fcode;
341     for (fcode = 1; (16 << fcode) <= MVmax; fcode++);
342     return fcode;
343 }
```

```
345 int xvid_me_SkipDecisionP(const IMAGE * current, const IMAGE * reference,
346     const int x, const int y, const uint32_t stride, const uint32_t iQuant)
347 {
348     int offset = (x + y*stride)*8;
349     uint32_t sadC = sad8(current->u + offset, reference->u + offset, stride);
350     if (sadC > iQuant * MAX_CHROMA_SAD_FOR_SKIP) return 0;

352     sadC += sad8(current->v + offset, reference->v + offset, stride);
353     if (sadC > iQuant * MAX_CHROMA_SAD_FOR_SKIP) return 0;

355     return 1;
356 }
```

/* 第 345 行到第 356 行是计算色度分量的 SAD 值, 因为根据人类对亮度相对敏感而对色度相对不敏感的视觉特性, 可以不计算色度分量的 SAD 值, 只用亮度分量判决。

```
// */
```

```
358 static __inline void get_pmvdata2(const MACROBLOCK * const pMB, const int mb_width,
359     const int x, const int y, VECTOR * const pmv, int32_t * const psad)
360 {
361     int num_cand = 0, last_cand = 1;

363     if (x)
364     {
365         num_cand++;
366         last_cand = 1;
367         pmv[1] = pMB[-1].mvs;
368         psad[1] = pMB[-1].sad8[1];
369     }
370     else
371     {
372         pmv[1] = zeroMV;
373         psad[1] = MV_MAX_ERROR;
374     }

376     if (y)
377     {
378         num_cand++;
379         last_cand = 2;
380         pmv[2] = pMB[-mb_width].mvs;
381         psad[2] = pMB[-mb_width].sad8[2];

383         if(x < mb_width - 1)
384         {
385             num_cand++;
386             last_cand = 3;
387             pmv[3] = pMB[1-mb_width].mvs;
388             psad[3] = pMB[1-mb_width].sad8[2];
389         }
390         else
391         {
392             pmv[3] = zeroMV;
393             psad[3] = MV_MAX_ERROR;
394         }
395     }
```

```
396     else
397     {
398         pmv[2] = zeroMV;
399         psad[2] = MV_MAX_ERROR;
400     }

402     if (x == 0 && y == 0)
403     {
404         pmv[0] = pmv[1] = pmv[2] = pmv[3] = zeroMV;
405         psad[0] = 0;
406         psad[1] = psad[2] = psad[3] = MV_MAX_ERROR;
407         return;
408     }

410     if (num_cand == 1)
411     {
412         pmv[0] = pmv[last_cand];
413         psad[0] = psad[last_cand];
414         return;
415     }

417     if ((MVequal(pmv[1], pmv[2])) && (MVequal(pmv[1], pmv[3])))
418     {
419         pmv[0] = pmv[1];
420         psad[0] = MIN(MIN(psad[1], psad[2]), psad[3]);
421         return;
422     }

424     pmv[0].x = MIN(MAX(pmv[1].x, pmv[2].x),
425                    MIN(MAX(pmv[2].x, pmv[3].x), MAX(pmv[1].x, pmv[3].x)));
426     pmv[0].y = MIN(MAX(pmv[1].y, pmv[2].y),
427                    MIN(MAX(pmv[2].y, pmv[3].y), MAX(pmv[1].y, pmv[3].y)));

429     psad[0] = MIN(MIN(psad[1], psad[2]), psad[3]);
430 }

/* 第 358 行到第 430 行根据 MP4 标准规定的中值预测计算运动矢量的预测值。主要是一些边界判断，候
   选运动矢量计算，最后选定中间值做为预测值。
// */
```

```
432 static void ModeDecision_SAD(SearchData* const Data, MACROBLOCK* const pMB,
```

```
433         const MACROBLOCK* const pMBs, const int x, const int y,
434         const MBParam * const pParam, const IMAGE * const pCurrent,
435         const IMAGE * const pRef, const int coding_type, const int skip_sad,
436         int * const MVmax, int * const mvCount, int * const mvSum)
437 {
438     const uint32_t iQuant = pParam->quant;

440     const int skip_possible = (coding_type == P_VOP);

442     int sad = Data->iMinSAD;
443     int InterBias = MV16_INTER_BIAS;

445     if (skip_possible && (skip_sad < (int)iQuant * MAX_SAD00_FOR_SKIP))
446         if ( (100*skip_sad)/(pMB->sad16+1) > FINAL_SKIP_THRESH)
447             if (xvid_me_SkipDecisionP(pCurrent, pRef, x, y, Data->iEdgedWidth/2, iQuant))
448                 {
449                     ZeroMacroblockP(pMB, 0);
450                     pMB->mode = MODE_NOT_CODED;

452                     return ;
453                 }

455     if (iQuant > 10)
456         InterBias += 60 * (iQuant - 10);

458     if (y != 0)
459         if ((pMB - pParam->mb_width)->mode == MODE_INTRA )
460             InterBias -= 80;

462     if (x != 0)
463         if ((pMB - 1)->mode == MODE_INTRA )
464             InterBias -= 80;

466     if (InterBias < sad)
467     {
468         int32_t deviation = dev16(Data->Cur, Data->iEdgedWidth);
469         if (deviation < (sad - InterBias))
470         {
471             ZeroMacroblockP(pMB, 0);
472             pMB->mode = MODE_INTRA;
```

```
474         return ;
475     }
476 }

478     pMB->cbp = 63;
479     pMB->sad16 = pMB->sad8[0] = pMB->sad8[1] = pMB->sad8[2] = pMB->sad8[3] = sad;

481     pMB->mvs = Data->currentMV;

483     pMB->pmvs.x = Data->currentMV.x - Data->predMV.x;
484     pMB->pmvs.y = Data->currentMV.y - Data->predMV.y;

486     pMB->mode = MODE_INTER;

488     {
489         const VECTOR * const mv = &(pMB->mvs);
490         int max = *MVmax;

492         (*mvCount)++;
493         *mvSum += mv[0].x * mv[0].x;
494         *mvSum += mv[0].y * mv[0].y;
495         if (mv[0].x > max) max = mv[0].x;
496         else if (-mv[0].x - 1 > max) max = -mv[0].x - 1;
497         if (mv[0].y > max) max = mv[0].y;
498         else if (-mv[0].y - 1 > max) max = -mv[0].y - 1;
499         *MVmax = max;
500     }
501 }
/* 第 432 行到第 501 行判决宏块的编码类型, 计算宏块参数的一些初始值, 对inter 类型还要计算 mvCount
   和 mvMax。
// */

503 static __inline void PreparePredictionsP(VECTOR * const pmv, int x, int y,
504     int iwcount, int ihcount, const MACROBLOCK * const prevMB)
505 {
506     if ( (y != 0) && (x < (iwcount-1)) ) // [5] top-right neighbour
507     {
508         pmv[5].x = EVEN(pmv[3].x);
509         pmv[5].y = EVEN(pmv[3].y);
```

```
510     }
511     else pmv[5].x = pmv[5].y = 0;

513     if (x != 0) // pmv[3] is left neighbour
514     {
515         pmv[3].x = EVEN(pmv[1].x);
516         pmv[3].y = EVEN(pmv[1].y);
517     }
518     else pmv[3].x = pmv[3].y = 0;

520     if (y != 0) // [4] top neighbour
521     {
522         pmv[4].x = EVEN(pmv[2].x);
523         pmv[4].y = EVEN(pmv[2].y);
524     }
525     else pmv[4].x = pmv[4].y = 0;

527     pmv[1].x = EVEN(pmv[0].x); // [1] median prediction
528     pmv[1].y = EVEN(pmv[0].y);

530     // [0] is zero;not used in the loop(changed before) but needed here for make_mask
531     pmv[0].x = pmv[0].y = 0;

533     pmv[2].x = EVEN(prevMB->mvs.x); // [2] is last frame
534     pmv[2].y = EVEN(prevMB->mvs.y);

536     if ((x < iWcount-1) && (y < iHcount-1))// [6] right-down neighbour in last frame
537     {
538         pmv[6].x = EVEN((prevMB+1+iWcount)->mvs.x);
539         pmv[6].y = EVEN((prevMB+1+iWcount)->mvs.y);
540     } else pmv[6].x = pmv[6].y = 0;
541 }

/* 第 503 行和第 541 行计算各个可能方向的运动矢量, 程序用这些运动矢量对应的 SAD 最小值来决定初始的预测起点, 即决定从那个点开始做整像素搜索和半像素搜索。
// */

543 static void SearchP(const IMAGE* const pRef, const uint8_t* const pRefH,
544                    const uint8_t* const pRefV, const uint8_t * const pRefHV,
545                    const IMAGE * const pCur, const int x, const int y,
546                    SearchData * const Data, const MBParam * const pParam,
```

```
547         const MACROBLOCK * const prevMB, MACROBLOCK * const pMB)
548 {
549     int threshA;
550     int i= (x + y * Data->iEdgedWidth) << 4;
551     VECTOR pmv[7];

553     get_range(&Data->min_dx, &Data->max_dx, &Data->min_dy, &Data->max_dy, x, y,
554             pParam->width, pParam->height, Data->iFcode);

556     get_pmvdata2(pMB, pParam->mb_width, x, y, pmv, Data->temp);

558     Data->Cur = pCur->y + i;

560     Data->RefP[0] = pRef->y + i;
561     Data->RefP[2] = pRefH + i;
562     Data->RefP[1] = pRefV + i;
563     Data->RefP[3] = pRefHV + i;

565     Data->lambda16 = xvid_me_lambda_vec16[pParam->quant];
566     Data->dir = 0;

568     Data->currentMV.x=Data->currentMV.y=0;
569     Data->predMV = pmv[0];

571     i = d_mv_bits(0, 0, Data->predMV, Data->iFcode);
572     Data->iMinSAD = pMB->sad16 + (Data->lambda16 * i);

574     if (x | y)
575     {
576         threshA = Data->temp[0]; /* that's where we keep this SAD atm */
577         if (threshA < 512) threshA = 512;
578         else if (threshA > 1024) threshA = 1024;
579     } else
580         threshA = 512;

582     PreparePredictionsP(pmv, x, y, pParam->mb_width, pParam->mb_height, prevMB);

584     for (i = 1; i < 7; i++)
585         if (!vector_repeats(pmv, i))
586             {
```

```
587         CheckCandidateVector(pmv[i].x, pmv[i].y, Data, i);
588         if (Data->iMinSAD <= threshA) { i++; break; }
589     }

591     if((Data->iMinSAD > threshA)
592         && ((Data->currentMV.x != prevMB->mvs.x) || (Data->currentMV.y != prevMB->mvs.y)
593             || (Data->iMinSAD >= prevMB->sad16)))
594     {
595         int mask = make_mask(pmv, i, Data->dir); //all vectors pmv[0..i-1] have been
checked

597         xvid_me_DiamondSearch(Data->currentMV.x, Data->currentMV.y, Data, mask);
598     }

600     xvid_me_SubpelRefine(Data->currentMV.x, Data->currentMV.y, Data);
601 }

/* 第 543 行到第 601 行是运动矢量搜索，第 553 行和第 554 行限定搜索范围，第 556 行计算运动矢量预
    测值，在计算预测偏移后第 582 行准备计算预测起点的各个方向上可能的运动矢量，在第 584 行到
    第 589 行决定预测起点，第 591 行到第 598 行是整像素搜索，第 600 行是半像素搜索。
// */

603 void MotionEstimation(MBParam* const pParam, FRAMEINFO* const current,
604     FRAMEINFO* const reference, const IMAGE* const pRefH,
605     const IMAGE* const pRefV, const IMAGE * const pRefHV)
606 {
607     MACROBLOCK *const pMBs = current->mbs;
608     const IMAGE *const pCurrent = &current->image;
609     const IMAGE *const pRef = &reference->image;

611     const uint32_t mb_width = pParam->mb_width;
612     const uint32_t mb_height = pParam->mb_height;
613     const uint32_t iEdgedWidth = pParam->edged_width;

615     const uint32_t iQuant = pParam->quant;

617     int MVmax = 0, mvSum = 0, mvCount = 0;

619     uint32_t x, y;
620     int32_t sad0;
621     int skip_thresh = INITIAL_SKIP_THRESH;
```

```
623     SearchData Data;
624     memset(&Data, 0, sizeof(SearchData));
625     Data.iEdgedWidth = iEdgedWidth;
626     Data.iFcode = current->fcode;
627     Data.rounding = pParam->m_rounding_type;

629     Data.RefQ = pRefV->u;

631     for (y = 0; y < mb_height; y++)
632     {
633         for (x = 0; x < mb_width; x++)
634         {
635             MACROBLOCK *pMB = &pMBs[x + y * pParam->mb_width];

637             pMB->sad16 = sad16v(pCurrent->y + (x + y * iEdgedWidth) * 16,
638                 pRef->y + (x + y * iEdgedWidth) * 16, pParam->edged_width, pMB->sad8);

640             sad0=4*MAX(MAX(pMB->sad8[0], pMB->sad8[1]), MAX(pMB->sad8[2], pMB->sad8[3]));

642             if (sad0 < pParam->quant * skip_thresh)
643                 if (xvid_me_SkipDecisionP(pCurrent, pRef, x, y, iEdgedWidth/2, iQuant))
644                 {
645                     ZeroMacroblockP(pMB, sad0);
646                     pMB->mode = MODE_NOT_CODED;
647                     continue;
648                 }

650             SearchP(pRef, pRefH->y, pRefV->y, pRefHV->y, pCurrent, x, y, &Data, pParam,
651                 reference->mbs + (x + y * pParam->mb_width), pMB);

653             ModeDecision_SAD(&Data, pMB, pMBs, x, y, pParam, pCurrent, pRef,
654                 current->coding_type, sad0, &MVmax, &mvCount, &mvSum);
655         }
656     }

658     current->fcode = getMinFcode(MVmax);
659     current->sStat.iMvSum = mvSum;
660     current->sStat.iMvCount = mvCount;
```



```
662     return ;
```





```
663 }
```

```
/* 第 603 行到第 663 行是完整的运动估计，第 650 行和第 651 行搜索最佳运动矢量，第 653 行和第 654  
   行判决宏块使用 intra 模式编码还是 inter 模式编码，并且记录相关参数，第 658 行到第 660 行记  
   录几个统计参数，用于计算自适应运动矢量的模。
```

```
// */
```

第五章 宏块级实用程序

5.1 文件列表

类型	名称	大小
	mem_align.h	801 bytes
	mem_transfer.h	8192 bytes
	mbfunctions.h	478 bytes
	mbfunctions.c	6455 bytes

5.2 mem_align.h 文件

5.2.1 功能描述

对奇内存分配和释放，目前 XVid 是 64 字节对齐。

5.2.2 文件注释

```
1  #ifndef _MEM_ALIGN_H_
2  #define _MEM_ALIGN_H_

4  static __inline void *xvid_malloc(int32_t size, uint8_t align)
5  {
6      uint8_t *mem_ptr;

8      if (!align)
9      {
10         if ((mem_ptr = (uint8_t *) malloc(size + 1)) != NULL)
11         {
12             *mem_ptr = (uint8_t)1;
13             return ((void *) (mem_ptr+1));
14         }
15     }
16     else
17     {
18         uint8_t *tmp;

20         if ((tmp = (uint8_t *) malloc(size + align)) != NULL)
21         {
22             mem_ptr = (uint8_t*)((uint32_t)(tmp+align-1) & ~(uint32_t)(align - 1));

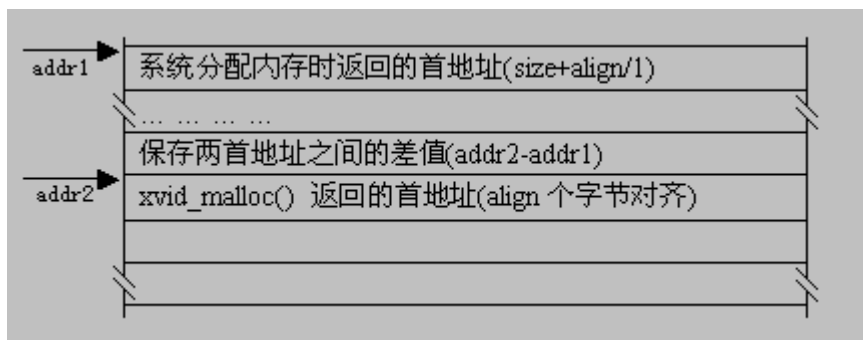
24             if (mem_ptr == tmp) // 处理起始地址刚好是内存对齐值的情况。
25                 mem_ptr += align;
```

```

27         *(mem_ptr - 1) = (uint8_t) (mem_ptr - tmp);

29         return ((void *)mem_ptr);
30     }
31 }
32 return(NULL);
33 }
/*

```



图十一 xvid_malloc() 函数分配时内存布局示意图

程序为了统一处理对齐内存分配和非对齐内存分配，统一对齐内存释放和非对齐内存释放，程序多分配一些字节，在返回给调用函数的指针前一个字节保存实际分配的内存指针到返回给调用函数的指针之间的偏移量。

第 10 行到第 13 行表示如果是非对齐内存分配，就多分配一个字节，起始地址字节保存偏移量 1，返回下一个地址值。

第 20 行到第 30 行表示如果是对齐内存分配，就多分配对齐的字节数，做逻辑运算保证内存对齐。第 22 行判断起始地址是否内存对齐值的，是就往下偏一个对齐字节值。第 27 行在对齐字节地址前一个字节地址保存偏移的字节数，返回对齐的地址值。

因为程序只有了一个字节来存储偏移量，所以最多只能是 256 字节对齐，超过 256 字节对齐的在最后释放内存时会出错。好在普通程序不会超过 256 字节对齐。

内存释放时，第 42 行首先强制转换程 uint8_t 类型，第 44 行取得字节偏移值，补偿偏移后得到正确的内存起始地址就可以直接释放内存了。

```

// */

35 static __inline void xvid_free(void *mem_ptr)
36 {
37     uint8_t *ptr;

39     if (mem_ptr == NULL)

```

```
40         return;

42     ptr = (uint8_t *)mem_ptr;

44     ptr -= *(ptr - 1);

46     free(ptr);
47 }

49 #endif
```

5.3 mem_transfer.h 文件

5.3.1 功能描述

宏块数据在帧缓冲区和核心缓冲区 dct_codes 之间的搬移，包括 C 语言和 PC 汇编语言版本，带 s 后缀的汇编宏定义表示是简单版(small)，最后一个调用，少几行汇编代码。

5.3.2 文件注释

```
1  #ifndef _MEM_TRANSFER_H
2  #define _MEM_TRANSFER_H

4  #ifdef TRANSFER_ACC

6  #define COPY_8_TO_16          \
7      __asm movq mm0, [eax]      \
8      __asm movq mm1, [eax+edx]  \
9      __asm movq mm2, mm0        \
10     __asm movq mm3, mm1         \
11     __asm punpcklbw mm0, mm7    \
12     __asm punpcklbw mm1, mm7    \
13     __asm punpckhbw mm2, mm7    \
14     __asm punpckhbw mm3, mm7    \
15     __asm movq [ecx], mm0       \
16     __asm movq [ecx+16], mm1    \
17     __asm movq [ecx+8], mm2     \
18     __asm movq [ecx+24], mm3    \
19     __asm lea  eax, [eax+2*edx]  \ // 源地址偏移两行
20     __asm add  ecx, 32           \ // 目的地址偏移 32 字节

22 #define COPY_8_TO_16s        \
23     __asm movq mm0, [eax]      \
24     __asm movq mm1, [eax+edx]  \
25     __asm movq mm2, mm0        \
```

```

26     __asm movq mm3, mm1      \
27     __asm punpcklbw mm0, mm7 \
28     __asm punpcklbw mm1, mm7 \
29     __asm punpckhbw mm2, mm7 \
30     __asm punpckhbw mm3, mm7 \
31     __asm movq [ecx], mm0    \
32     __asm movq [ecx+16], mm1 \
33     __asm movq [ecx+8], mm2  \
34     __asm movq [ecx+24], mm3

```

```

36 static __inline void transfer_8tol6copy(int16_t* const dst, const uint8_t* const src,
37                                     uint32_t stride)
38 {
39     __asm mov ecx, dst ; Dst
40     __asm mov eax, src ; Src
41     __asm mov edx, stride ; Stride

43     __asm pxor mm7, mm7

45     COPY_8_TO_16
46     COPY_8_TO_16
47     COPY_8_TO_16
48     COPY_8_TO_16s
49 }

```

/* 第 36 行到第 49 行实现从 8x8 小块无符号 8bit 源拷贝到 8x8 无符号 16bit 目的地址中，主要是利用 punpcklbw 和 punpckhbw 指令做 8bit 到 16bit 无符号数转换。

应用于编码 Intra 宏块时从图像缓冲区中转换拷贝一个宏块的数据到核心数组 data[] 中，所以 dst 和 src 的 stride 不同。

// */

```

51 #define COPY_8_TO_8      \
52     __asm movq mm0, [eax] \
53     __asm movq mm1, [eax+edx] \
54     __asm movq [ecx], mm0    \
55     __asm movq [ecx+edx], mm1 \
56     __asm lea eax, [eax+2*edx] \
57     __asm lea ecx, [ecx+2*edx]

59 #define COPY_8_TO_8s      \
60     __asm movq mm0, [eax] \
61     __asm movq mm1, [eax+edx] \
62     __asm movq [ecx], mm0    \
63     __asm movq [ecx+edx], mm1

```

```

65 static __inline void transfer8x8_copy(uint8_t* const dst, const uint8_t* const src,
66                                     const uint32_t stride)

```

```
67 {
68     __asm mov ecx, dst ; Dst
69     __asm mov eax, src ; Src
70     __asm mov edx, stride ; Stride
```

```
72     COPY_8_TO_8
73     COPY_8_TO_8
74     COPY_8_TO_8
75     COPY_8_TO_8s
76 }
```

/* 第65行到第76行利用mm0和mm1中转实现从8x8小块无符号8bit源拷贝到8x8无符号8bit目的地中，图像之间直接拷贝，所以dst和src的stride相同。

应用于MODE_NOT_CODED宏块从参考图像缓冲区中拷贝一个宏块的色度分量数据到当前图像缓冲区作为下帧的预测值。

```
// */
```

```
78 #define COPY_16_TO_16 \
79     __asm movdqa xmm0, [eax] \
80     __asm movdqa xmm1, [eax+edx] \
81     __asm movdqa [ecx], xmm0 \
82     __asm movdqa [ecx+edx], xmm1 \
83     __asm lea eax, [eax+2*edx] \
84     __asm lea ecx, [ecx+2*edx]
```

```
86 #define COPY_16_TO_16s \
87     __asm movdqa xmm0, [eax] \
88     __asm movdqa xmm1, [eax+edx] \
89     __asm movdqa [ecx], xmm0 \
90     __asm movdqa [ecx+edx], xmm1
```

```
92 static __inline void transfer16x16_copy(uint8_t* const dst, const uint8_t* const src,
93     const uint32_t stride)
```

```
94 {
95     __asm mov ecx, dst ; Dst
96     __asm mov eax, src ; Src
97     __asm mov edx, stride ; Stride
```

```
99     COPY_16_TO_16
100    COPY_16_TO_16
101    COPY_16_TO_16
102    COPY_16_TO_16
103    COPY_16_TO_16
104    COPY_16_TO_16
105    COPY_16_TO_16
106    COPY_16_TO_16s
107 }
```

/* 第 92 行到第 107 行利用 xmm0 和 xmm1 中转实现从 16x16 宏块无符号 8bit 源拷贝到 16x16 无符号 8bit 目的地址中，图像之间直接拷贝，所以 dst 和 src 的 stride 相同。

此函数用于 MODE_NOT_CODED 宏块从参考图像缓冲区中拷贝一个宏块的亮度分量数据到当前图像缓冲区，作为下帧的预测值。

// */

```
109 #define COPY_16_T0_8          \
110     __asm movdqa xmm0, [eax]    \
111     __asm movdqa xmm1, [eax+16] \
112     __asm packuswb xmm0, xmm7   \
113     __asm packuswb xmm1, xmm7   \
114     __asm movlps [ecx], xmm0     \
115     __asm movlps [ecx+edx], xmm1 \
116     __asm add eax, 32            \
117     __asm lea ecx, [ecx+2*edx]
```

```
119 #define COPY_16_T0_8s         \
120     __asm movdqa xmm0, [eax]    \
121     __asm movdqa xmm1, [eax+16] \
122     __asm packuswb xmm0, xmm7   \
123     __asm packuswb xmm1, xmm7   \
124     __asm movlps [ecx], xmm0     \
125     __asm movlps [ecx+edx], xmm1
```

```
127 static __inline void transfer_16to8copy(uint8_t* const dst, const int16_t* const src,
128     uint32_t stride)
```

```
129 {
```

```
130     __asm mov ecx, dst
131     __asm mov eax, src
132     __asm mov edx, stride
```

```
134     __asm pxor xmm7, xmm7
```

```
136     COPY_16_T0_8
137     COPY_16_T0_8
138     COPY_16_T0_8
139     COPY_16_T0_8s
140 }
```

/* 第 127 行到第 140 行实现从 8x8 小块无符号 16bit 源拷贝到 8x8 无符号 8bit 目的地址中，主要是利用 packuswb 指令做 16bit 到 8bit 无符号数转换。

此函数应于编码 intra 宏块时从核心数组 data[] 中转换拷贝一个宏块的重构数据到图像缓冲区中，所以 dst 和 src 的 stride 不同。

// */

```
142 #define COPY_16_T0_8_ADD      \
143     __asm movlps xmm0, [ecx]    \
```

```

144     __asm movlps xmm2, [ecx+edx]    \
145     __asm punpcklbw xmm0, xmm7     \
146     __asm punpcklbw xmm2, xmm7     \
147     __asm paddsw xmm0, [eax]       \
148     __asm paddsw xmm2, [eax+16]    \
149     __asm packuswb xmm0, xmm7      \
150     __asm packuswb xmm2, xmm7      \
151     __asm movlps [ecx], xmm0       \
152     __asm movlps [ecx+edx], xmm2   \
153     __asm lea ecx, [ecx+2*edx]      \
154     __asm add eax, 32

156 #define COPY_16_TO_8_ADDs         \
157     __asm movlps xmm0, [ecx]       \
158     __asm movlps xmm2, [ecx+edx]   \
159     __asm punpcklbw xmm0, xmm7     \
160     __asm punpcklbw xmm2, xmm7     \
161     __asm paddsw xmm0, [eax]       \
162     __asm paddsw xmm2, [eax+16]    \
163     __asm packuswb xmm0, xmm7      \
164     __asm packuswb xmm2, xmm7      \
165     __asm movlps [ecx], xmm0       \
166     __asm movlps [ecx+edx], xmm2

168 static __inline void transfer_16to8add(uint8_t* const dst, const int16_t* const src,
169                                         uint32_t stride)
170 {
171     __asm mov ecx, dst
172     __asm mov eax, src
173     __asm mov edx, stride

175     __asm pxor xmm7, xmm7

177     COPY_16_TO_8_ADD
178     COPY_16_TO_8_ADD
179     COPY_16_TO_8_ADD
180     COPY_16_TO_8_ADDs
181 }
/* 第 168 行到第 181 行实现从 8x8 小块无符号 16bit 源数据加到 8x8 无符号 8bit 目的地址数据中，主
   要流程是利用 punpcklbw 做目的地址数据 8bit 到 16bit 扩展，利用 paddsw 做饱和加法，利用
   packuswb 指令做 16bit 到 8bit 无符号数转换。
   此函数用于编码 inter 宏块时修正图像数据预测值，数据从 data[] 数组加到图像缓冲区中，所以 dst
   和 src 的 stride 不同。
// */

183 #define COPY_8_TO_16_SUB          \

```



```
184     __asm movlps xmm0, [ebx]          \
185     __asm movlps xmm1, [ebx+edx]      \
186     __asm movlps xmm2, [eax]          \
187     __asm movlps xmm3, [eax+edx]      \
188     __asm movlps [eax], xmm0           \
189     __asm movlps [eax+edx], xmm1       \
190                                     \
191     __asm punpcklbw xmm0, xmm7         \
192     __asm punpcklbw xmm1, xmm7         \
193     __asm punpcklbw xmm2, xmm7         \
194     __asm punpcklbw xmm3, xmm7         \
195                                     \
196     __asm psubw xmm2, xmm0             \
197     __asm psubw xmm3, xmm1             \
198                                     \
199     __asm movdqu [ecx], xmm2           \
200     __asm movdqu [ecx+16], xmm3        \
201                                     \
202     __asm lea ebx, [ebx+2*edx]          \
203     __asm lea eax, [eax+2*edx]          \
204     __asm add ecx, 32

206 #define COPY_8_TO_16_SUBs             \
207     __asm movlps xmm0, [ebx]           \
208     __asm movlps xmm1, [ebx+edx]        \
209     __asm movlps xmm2, [eax]           \
210     __asm movlps xmm3, [eax+edx]        \
211     __asm movlps [eax], xmm0            \
212     __asm movlps [eax+edx], xmm1        \
213                                     \
214     __asm punpcklbw xmm0, xmm7          \
215     __asm punpcklbw xmm1, xmm7          \
216     __asm punpcklbw xmm2, xmm7          \
217     __asm punpcklbw xmm3, xmm7          \
218                                     \
219     __asm psubw xmm2, xmm0              \
220     __asm psubw xmm3, xmm1              \
221                                     \
222     __asm movdqu [ecx], xmm2            \
223     __asm movdqu [ecx+16], xmm3

225 static __inline void transfer_8to16sub(int16_t* const dst, uint8_t* const cur,
226                                     const uint8_t* ref, const uint32_t stride)
227 {
228     __asm mov ecx, dst
229     __asm mov eax, cur
```

```
230     __asm mov edx, stride
231     __asm mov ebx, ref

233     __asm pxor xmm7, xmm7
```

```
235     COPY_8_TO_16_SUB
236     COPY_8_TO_16_SUB
237     COPY_8_TO_16_SUB
238     COPY_8_TO_16_SUBs
239 }
```

/* 第 225 行到第 239 行实现从 8x8 小块无符号 8bit 图像数据源中减去预测值，把结果转换成 16 比特保存到 data[] 核心数组中。

第 188 行到第 189 行和第 211 行到第 212 行是把参考图像数据值拷贝到当前编码图像相应位置中作为预测值的一部分。

预测值的另一部分是重构的图像数据值，在 transfer_16to8add() 函数中加上来。

此函数用于编码 inter 宏块时做运动补偿，数据从当前图像缓冲区中减参考图像数据后到 data[] 核心数组中，所以 dst 和 src 的 stride 不同。

```
// */
```

```
241 static __inline void transfer16x16_8tol6sub(int16_t* dst, uint8_t* const cur,
242         const uint8_t* ref, const uint32_t stride)
243 {
244     int stride8 = stride << 3;

246     transfer_8tol6sub(dst, cur, ref, stride);
247     transfer_8tol6sub(dst+64, cur+8, ref+8, stride);
248     transfer_8tol6sub(dst+128, cur+stride8, ref+stride8, stride);
249     transfer_8tol6sub(dst+192, cur+stride8+8, ref+stride8+8, stride);
250 }
```

```
253 #else
```

```
255 static __inline void transfer_8tol6copy(int16_t* const dst, const uint8_t* const src,
256         uint32_t stride)
257 {
258     uint32_t i, j;

260     for (j = 0; j < 8; j++)
261     {
262         for (i = 0; i < 8; i++)
263             dst[j * 8 + i] = (int16_t) src[j * stride + i];
264     }
265 }
```

/* 第 255 行到第 265 行应用于编码 Intra 宏块时从图像缓冲区中转换拷贝一个宏块的数据到核心数组 data[] 中。

```
// */
```

```
267 static __inline void transfer_16to8copy(uint8_t* const dst, const int16_t* const src,
268         uint32_t stride)
269 {
270     uint32_t i, j;

272     for (j = 0; j < 8; j++)
273     {
274         for (i = 0; i < 8; i++)
275         {
276             int16_t pixel = src[j * 8 + i];

278             if (pixel < 0)
279             {
280                 pixel = 0;
281             }
282             else if (pixel > 255)
283             {
284                 pixel = 255;
285             }
286             dst[j * stride + i] = (uint8_t) pixel;
287         }
288     }
289 }
/* 第 267 行到 289 行用于编码 intra 宏块时从核心数组 data[] 中转换拷贝一个宏块的重构数据到图像缓
   冲区中，作为下一帧的预测值。
// */

291 static __inline void transfer_8to16sub(int16_t* const dct, uint8_t* const cur,
292         const uint8_t* ref, const uint32_t stride)
293 {
294     uint32_t i, j;

296     for (j = 0; j < 8; j++)
297     {
298         for (i = 0; i < 8; i++)
299         {
300             uint8_t c = cur[j * stride + i];
301             uint8_t r = ref[j * stride + i];

303             cur[j * stride + i] = r;
304             dct[j * 8 + i] = (int16_t) c - (int16_t) r;
305         }
306     }
307 }
/* 第 291 行到第 307 行应用于编码 inter 宏块时做运动补偿，数据从当前图像缓冲区中减参考图像数据
```

后到 data[] 核心数组中

/* 特别注意第 303 行, 拷贝参考图像数据值作为下一帧的预测值的一部分, 另一部分是重构的图像数据值在 transfer_16to8add() 函数中加上来。

// */

```
309 static __inline void transfer_16to8add(uint8_t* const dst, const int16_t* const src,
310                                         uint32_t stride)
311 {
312     uint32_t i, j;

314     for (j = 0; j < 8; j++)
315     {
316         for (i = 0; i < 8; i++)
317         {
318             int16_t pixel = (int16_t) dst[j * stride + i] + src[j * 8 + i];

320             if (pixel < 0)
321             {
322                 pixel = 0;
323             }
324             else if (pixel > 255)
325             {
326                 pixel = 255;
327             }
328             dst[j * stride + i] = (uint8_t) pixel;
329         }
330     }
331 }
```

/* 第 309 行到第 331 行把 data[] 数组中重构的 inter 宏块残差数据值加到图像缓冲区中, 这样就是一个完整的预测值, 用于下一帧图像编码的预测值。

// */

```
333 static __inline void transfer8x8_copy(uint8_t* const dst, const uint8_t* const src,
334                                         const uint32_t stride)
335 {
336     uint32_t j;

338     for (j = 0; j < 8; j++)
339     {
340         uint32_t *d= (uint32_t*)(dst + j*stride);
341         const uint32_t *s = (const uint32_t*)(src + j*stride);
342         *(d+0) = *(s+0);
343         *(d+1) = *(s+1);
344     }
345 }
```

// 用于 MODE_NOT_CODED 宏块从参考图像缓冲区中拷贝一个宏块的色度分量数据到当前图像缓冲区

```
347 static __inline void transfer16x16_copy(uint8_t* const dst, const uint8_t* const src,
```

```
348         const uint32_t stride)
349 {
350     transfer8x8_copy(dst, src, stride);
351     transfer8x8_copy(dst + 8, src + 8, stride);
352     transfer8x8_copy(dst + 8*stride, src + 8*stride, stride);
353     transfer8x8_copy(dst + 8*stride + 8, src + 8*stride + 8, stride);
354 }
// 应用于 MODE_NOT_CODED 宏块从参考图像缓冲区中拷贝一个宏块的亮度分量数据到当前图像缓冲区

356 #endif
```

5.4 mbfunctions.h 文件

5.4.1 功能描述

Intra 宏块及 Inter 宏块前向计算和重构计算共两个函数原型的声明。

5.4.2 文件注释

```
1  #ifndef _ENCORE_BLOCK_H
2  #define _ENCORE_BLOCK_H

4  void MBTransQuantIntra(const MBParam* const pParam, const FRAMEINFO* const frame,
5                          MACROBLOCK* const pMB, const uint32_t x_pos, const uint32_t y_pos,
6                          int16_t data[6 * 64], int16_t qcoeff[6 * 64]);

8  uint8_t MBTransQuantInter(const MBParam* const pParam, const FRAMEINFO* const frame,
9                             MACROBLOCK * const pMB, const uint32_t x_pos, const uint32_t y_pos,
10                             int16_t data[6 * 64], int16_t qcoeff[6 * 64]);

12 #endif
```

5.5 mbfunctions.c 文件

5.5.1 功能描述

Intra 宏块及 Inter 宏块前向计算和重构计算共两函数的实现及其辅助函数的实现。

5.5.2 文件注释

```
1  #include "../portab.h"
2  #include "../global.h"
3  #include "../encoder.h"

5  #include "../dct/fdct.h"
```

```
6  #include "../dct/idct.h"
7  #include "../quant/quant.h"

9  #include "mbfunctions.h"
10 #include "mem_transfer.h"

12 static __inline void MBfDCT(int16_t data[6 * 64]) // 前向 DCT 变换
13 {
14     fdct(&data[0 * 64]);
15     fdct(&data[1 * 64]);
16     fdct(&data[2 * 64]);
17     fdct(&data[3 * 64]);
18     fdct(&data[4 * 64]);
19     fdct(&data[5 * 64]);
20 }

22 static __inline void MBTrans8to16(const MBParam* const pParam,
23     const FRAMEINFO* const frame, const MACROBLOCK * const pMB,
24     const uint32_t x_pos, const uint32_t y_pos, int16_t data[6 * 64])
25 {
26     const uint32_t stride = pParam->edged_width;
27     const uint32_t stride2 = stride / 2;
28     const uint32_t next_block = stride * 8;

30     const IMAGE * const pCurrent = &frame->image;

32     uint8_t* const pY_Cur = pCurrent->y + (y_pos << 4) * stride + (x_pos << 4);
33     uint8_t* const pU_Cur = pCurrent->u + (y_pos << 3) * stride2 + (x_pos << 3);
34     uint8_t* const pV_Cur = pCurrent->v + (y_pos << 3) * stride2 + (x_pos << 3);

36     transfer_8to16copy(&data[0 * 64], pY_Cur, stride);
37     transfer_8to16copy(&data[1 * 64], pY_Cur + 8, stride);
38     transfer_8to16copy(&data[2 * 64], pY_Cur + next_block, stride);
39     transfer_8to16copy(&data[3 * 64], pY_Cur + next_block + 8, stride);
40     transfer_8to16copy(&data[4 * 64], pU_Cur, stride2);
41     transfer_8to16copy(&data[5 * 64], pV_Cur, stride2);
42 }
/*
    第 36 行到第 41 行从图像缓冲区中拷贝一个宏块的数据到 data[] 核心数据缓冲区，因为 DCT 变换
    可能会超出 8bit 范围，所以要转换成 16bit 数。
    注意坐标位置不要算错。
*/
// */

44 static __inline void MBTrans16to8_0(const MBParam* const pParam,
45     const FRAMEINFO* const frame, const MACROBLOCK* const pMB,
46     const uint32_t x_pos, const uint32_t y_pos, int16_t data[6 * 64],
```

```

47         const uint8_t cbp)
48     {
49         const uint32_t stride = pParam->edged_width;
50         const uint32_t stride2 = stride / 2;
51         const uint32_t next_block = stride * 8;

53         const IMAGE * const pCurrent = &frame->image;

55         uint8_t* const pY_Cur = pCurrent->y + (y_pos << 4) * stride + (x_pos << 4);
56         uint8_t* const pU_Cur = pCurrent->u + (y_pos << 3) * stride2 + (x_pos << 3);
57         uint8_t* const pV_Cur = pCurrent->v + (y_pos << 3) * stride2 + (x_pos << 3);

59         transfer_16to8copy(pY_Cur,                &data[0 * 64], stride);
60         transfer_16to8copy(pY_Cur + 8,            &data[1 * 64], stride);
61         transfer_16to8copy(pY_Cur + next_block,    &data[2 * 64], stride);
62         transfer_16to8copy(pY_Cur + next_block + 8, &data[3 * 64], stride);
63         transfer_16to8copy(pU_Cur,                &data[4 * 64], stride2);
64         transfer_16to8copy(pV_Cur,                &data[5 * 64], stride2);
65     }
    /*

```

第44行到第65行把重构的一个 Intra 宏块的图像数据从 data[] 核心数据缓冲区拷贝到图像缓存区作为下一帧的预测值。

因为前面使用 16bit 数据，图像缓冲区使用 8bit 数据，所以要饱和成 8bit 数。

注意坐标位置不要算错。

// */

```

67 static __inline void MBTrans16to8_1(const MBParam* const pParam,
68         const FRAMEINFO* const frame, const MACROBLOCK * const pMB,
69         const uint32_t x_pos, const uint32_t y_pos, int16_t data[6 * 64],
70         const uint8_t cbp)
71     {
72         const uint32_t stride = pParam->edged_width;
73         const uint32_t stride2 = stride / 2;
74         const uint32_t next_block = stride * 8;

76         const IMAGE * const pCurrent = &frame->image;

78         uint8_t* const pY_Cur = pCurrent->y + (y_pos << 4) * stride + (x_pos << 4);
79         uint8_t* const pU_Cur = pCurrent->u + (y_pos << 3) * stride2 + (x_pos << 3);
80         uint8_t* const pV_Cur = pCurrent->v + (y_pos << 3) * stride2 + (x_pos << 3);

82         if (cbp & 32) transfer_16to8add(pY_Cur,                &data[0 * 64], stride);
83         if (cbp & 16) transfer_16to8add(pY_Cur + 8,            &data[1 * 64], stride);
84         if (cbp & 8)  transfer_16to8add(pY_Cur + next_block,    &data[2 * 64], stride);
85         if (cbp & 4)  transfer_16to8add(pY_Cur + next_block + 8, &data[3 * 64], stride);
86         if (cbp & 2)  transfer_16to8add(pU_Cur,                &data[4 * 64], stride2);
87         if (cbp & 1)  transfer_16to8add(pV_Cur,                &data[5 * 64], stride2);

```

```
88 }
/* 第 67 行到第 88 行把重构的一个 Inter 宏块的图像数据加上相应的预测值回写到图像缓存区作为下一
   帧的预测值。
   因为前面使用 16bit 数据，图像缓冲区使用 8bit 数据，所以要饱和成 8bit 数。
   为节省计算时间，先判断一下 cbp 来决定是否做加法和回写动作。
   注意坐标位置不要算错。
// */

90 void MBTransQuantIntra(const MBParam* const pParam, const FRAMEINFO* const frame,
91     MACROBLOCK * const pMB, const uint32_t x_pos, const uint32_t y_pos,
92     int16_t data[6 * 64], int16_t qcoeff[6 * 64])
93 {
94     const int iQuant = pParam->quant;
95     const int scaler_lum = frame->y_scale;
96     const int scaler_chr = frame->uv_scale;

98     MBTrans8to16(pParam, frame, pMB, x_pos, y_pos, data);

100     MBfDCT(data);

102     quant_h263_intra(&qcoeff[0 * 64], &data[0 * 64], iQuant, scaler_lum);
103     quant_h263_intra(&qcoeff[1 * 64], &data[1 * 64], iQuant, scaler_lum);
104     quant_h263_intra(&qcoeff[2 * 64], &data[2 * 64], iQuant, scaler_lum);
105     quant_h263_intra(&qcoeff[3 * 64], &data[3 * 64], iQuant, scaler_lum);
106     quant_h263_intra(&qcoeff[4 * 64], &data[4 * 64], iQuant, scaler_chr);
107     quant_h263_intra(&qcoeff[5 * 64], &data[5 * 64], iQuant, scaler_chr);

109     dequant_h263_intra(&data[0 * 64], &qcoeff[0 * 64], iQuant, scaler_lum);
110     dequant_h263_intra(&data[1 * 64], &qcoeff[1 * 64], iQuant, scaler_lum);
111     dequant_h263_intra(&data[2 * 64], &qcoeff[2 * 64], iQuant, scaler_lum);
112     dequant_h263_intra(&data[3 * 64], &qcoeff[3 * 64], iQuant, scaler_lum);
113     dequant_h263_intra(&data[4 * 64], &qcoeff[4 * 64], iQuant, scaler_chr);
114     dequant_h263_intra(&data[5 * 64], &qcoeff[5 * 64], iQuant, scaler_chr);

116     idct(&data[0 * 64]);
117     idct(&data[1 * 64]);
118     idct(&data[2 * 64]);
119     idct(&data[3 * 64]);
120     idct(&data[4 * 64]);
121     idct(&data[5 * 64]);

123     MBTrans16to8_0(pParam, frame, pMB, x_pos, y_pos, data, 0x3F);
124 }
/* 第 90 行到第 124 行是 intra 宏块前向计算和重构计算流程。
   大概流程是 16x16 宏块数据拷贝，dct 变换，量化，反量化，反 DCT 变换，回写重构图像。
// */
```



```
126 uint8_t MBTransQuantInter(const MBParam* const pParam, const FRAMEINFO* const frame,
127     MACROBLOCK* const pMB, const uint32_t x_pos, const uint32_t y_pos,
128     int16_t data[6 * 64], int16_t qcoeff[6 * 64])
129 {
130     uint8_t cbp=0;

132     const int iQuant = pParam->quant;

134     MBfDCT(data);

136     if(quant_h263_inter(&qcoeff[0*64], &data[0*64], iQuant))
137         cbp |= 32;
138     if(quant_h263_inter(&qcoeff[1*64], &data[1*64], iQuant))
139         cbp |= 16;
140     if(quant_h263_inter(&qcoeff[2*64], &data[2*64], iQuant))
141         cbp |= 8;
142     if(quant_h263_inter(&qcoeff[3*64], &data[3*64], iQuant))
143         cbp |= 4;
144     if(quant_h263_inter(&qcoeff[4*64], &data[4*64], iQuant))
145         cbp |= 2;
146     if(quant_h263_inter(&qcoeff[5*64], &data[5*64], iQuant))
147         cbp |= 1;

149     if(cbp & 32) dequant_h263_inter(&data[0 * 64], &qcoeff[0 * 64], iQuant);
150     if(cbp & 16) dequant_h263_inter(&data[1 * 64], &qcoeff[1 * 64], iQuant);
151     if(cbp & 8) dequant_h263_inter(&data[2 * 64], &qcoeff[2 * 64], iQuant);
152     if(cbp & 4) dequant_h263_inter(&data[3 * 64], &qcoeff[3 * 64], iQuant);
153     if(cbp & 2) dequant_h263_inter(&data[4 * 64], &qcoeff[4 * 64], iQuant);
154     if(cbp & 1) dequant_h263_inter(&data[5 * 64], &qcoeff[5 * 64], iQuant);

156     if(cbp & 32) idct(&data[0 * 64]);
157     if(cbp & 16) idct(&data[1 * 64]);
158     if(cbp & 8) idct(&data[2 * 64]);
159     if(cbp & 4) idct(&data[3 * 64]);
160     if(cbp & 2) idct(&data[4 * 64]);
161     if(cbp & 1) idct(&data[5 * 64]);

163     MBTrans16to8_1(pParam, frame, pMB, x_pos, y_pos, data, cbp);

165     return(cbp);
166 }
/* 第 126 行到第 166 行是 inter 宏块前向计算和重构计算流程。
   大致流程是 dct 变换，量化，反量化，反 DCT 变换，重构 16x16 宏块图像。
   注意 16x16 宏块数据拷贝是在 MBMotionCompensation() 函数中做的。
// */
```

第六章 DCT 变换

6.1 文件列表

类型	名称	大小
	fdct.h	80 bytes
	fdct.c	19511 bytes
	idct.h	107 bytes
	idct.c	13983 bytes

6.2 fdct.h 文件

6.2.1 功能描述

前向 8x8 二维 DCT 变换函数原型的声明。

6.2.2 文件注释

```
1  #ifndef _FDCT_H_
2  #define _FDCT_H_

4  void fdct(short *const block);

6  #endif
```

6.3 fdct.c 文件

6.3.1 功能描述

前向 8x8 二维 DCT 变换函数实现，请查阅其他相关书籍，包括 C 语言和 PC 汇编语言版本。

6.3.2 文件注释

```
1  #include "../portab.h"
2  #include "fdct.h"

4  #ifdef FDCT_ACC

6  #pragma warning( disable : 4305)

8  __declspec(align(16)) short tan1[8]={
9      0x32ec, 0x32ec, 0x32ec, 0x32ec,
10     0x32ec, 0x32ec, 0x32ec, 0x32ec};

12  __declspec(align(16)) short tan2[8]={
13     0x6a0a, 0x6a0a, 0x6a0a, 0x6a0a,
14     0x6a0a, 0x6a0a, 0x6a0a, 0x6a0a};
```

```
16 __declspec(aligned(16)) short tan3[8]={
17     0xab0e, 0xab0e, 0xab0e, 0xab0e,
18     0xab0e, 0xab0e, 0xab0e, 0xab0e};

20 __declspec(aligned(16)) short sqrt2[8]={
21     0x5a82, 0x5a82, 0x5a82, 0x5a82,
22     0x5a82, 0x5a82, 0x5a82, 0x5a82};

24 __declspec(aligned(16)) short fTab1[]={
25     0x4000, 0x4000, 0x58c5, 0x4b42,
26     0xdd5d, 0xac61, 0xa73b, 0xcdb7,
27     0x4000, 0x4000, 0x3249, 0x11a8,
28     0x539f, 0x22a3, 0x4b42, 0xee58,
29     0x4000, 0xc000, 0x3249, 0xa73b,
30     0x539f, 0xdd5d, 0x4b42, 0xa73b,
31     0xc000, 0x4000, 0x11a8, 0x4b42,
32     0x22a3, 0xac61, 0x11a8, 0xcdb7};

34 __declspec(aligned(16)) short fTab2[]={
35     0x58c5, 0x58c5, 0x7b21, 0x6862,
36     0xcff5, 0x8c04, 0x84df, 0xba41,
37     0x58c5, 0x58c5, 0x45bf, 0x187e,
38     0x73fc, 0x300b, 0x6862, 0xe782,
39     0x58c5, 0xa73b, 0x45bf, 0x84df,
40     0x73fc, 0xcff5, 0x6862, 0x84df,
41     0xa73b, 0x58c5, 0x187e, 0x6862,
42     0x300b, 0x8c04, 0x187e, 0xba41};

44 __declspec(aligned(16)) short fTab3[]={
45     0x539f, 0x539f, 0x73fc, 0x6254,
46     0xd2bf, 0x92bf, 0x8c04, 0xbe4d,
47     0x539f, 0x539f, 0x41b3, 0x1712,
48     0x6d41, 0x2d41, 0x6254, 0xe8ee,
49     0x539f, 0xac61, 0x41b3, 0x8c04,
50     0x6d41, 0xd2bf, 0x6254, 0x8c04,
51     0xac61, 0x539f, 0x1712, 0x6254,
52     0x2d41, 0x92bf, 0x1712, 0xbe4d};

54 __declspec(aligned(16)) short fTab4[]={
55     0x4b42, 0x4b42, 0x6862, 0x587e,
56     0xd746, 0x9dac, 0x979e, 0xc4df,
57     0x4b42, 0x4b42, 0x3b21, 0x14c3,
58     0x6254, 0x28ba, 0x587e, 0xeb3d,
59     0x4b42, 0xb4be, 0x3b21, 0x979e,
60     0x6254, 0xd746, 0x587e, 0x979e,
61     0xb4be, 0x4b42, 0x14c3, 0x587e,
```

```
62     0x28ba, 0x9dac, 0x14c3, 0xc4df};

64     __declspec(aligned(16)) short Fdct_Rnd0[]={ 6,8,8,8, 6,8,8,8};
65     __declspec(aligned(16)) short Fdct_Rnd1[]={ 8,8,8,8, 8,8,8,8};
66     __declspec(aligned(16)) short Fdct_Rnd2[]={ 10,8,8,8, 8,8,8,8};
67     __declspec(aligned(16)) short Rounder1[]={ 1,1,1,1, 1,1,1,1};

69 void fdct(short *const block)
70 {
71     __asm
72     {
73         mov ecx, block

75         //; fLLM_PASS ecx+0, 3
76         movdqa xmm0, [ecx+0*16] ; In0
77         movdqa xmm2, [ecx+2*16] ; In2
78         movdqa xmm3, xmm0
79         movdqa xmm4, xmm2
80         movdqa xmm7, [ecx+7*16] ; In7
81         movdqa xmm5, [ecx+5*16] ; In5

83         psubsw xmm0, xmm7 ; t7 = In0-In7
84         paddsw xmm7, xmm3 ; t0 = In0+In7
85         psubsw xmm2, xmm5 ; t5 = In2-In5
86         paddsw xmm5, xmm4 ; t2 = In2+In5

88         movdqa xmm3, [ecx+3*16] ; In3
89         movdqa xmm4, [ecx+4*16] ; In4
90         movdqa xmm1, xmm3
91         psubsw xmm3, xmm4 ; t4 = In3-In4
92         paddsw xmm4, xmm1 ; t3 = In3+In4
93         movdqa xmm6, [ecx+6*16] ; In6
94         movdqa xmm1, [ecx+1*16] ; In1
95         psubsw xmm1, xmm6 ; t6 = In1-In6
96         paddsw xmm6, [ecx+1*16] ; t1 = In1+In6

98         psubsw xmm7, xmm4 ; tm03 = t0-t3
99         psubsw xmm6, xmm5 ; tm12 = t1-t2
100        paddsw xmm4, xmm4 ; 2. t3
101        paddsw xmm5, xmm5 ; 2. t2
102        paddsw xmm4, xmm7 ; tp03 = t0+t3
103        paddsw xmm5, xmm6 ; tp12 = t1+t2

105        psllw xmm2, 4 ; shift t5 (shift +1 to..
106        psllw xmm1, 4 ; shift t6 ..compensate cos4/2)
107        psllw xmm4, 3 ; shift t3
```

```
108      psllw  xmm5, 3          ; shift t2
109      psllw  xmm7, 3          ; shift t0
110      psllw  xmm6, 3          ; shift t1
111      psllw  xmm3, 3          ; shift t4
112      psllw  xmm0, 3          ; shift t7

114      psubsw xmm4, xmm5        ; out4 = tp03-tp12
115      psubsw xmm1, xmm2        ; xmm1: t6-t5
116      paddsw xmm5, xmm5
117      paddsw xmm2, xmm2
118      paddsw xmm5, xmm4        ; out0 = tp03+tp12
119      movdqa [ecx+4*16], xmm4  ; => out4
120      paddsw xmm2, xmm1        ; xmm2: t6+t5
121      movdqa [ecx+0*16], xmm5  ; => out0

123      movdqa xmm4, [tan2]     ; xmm4 <= tan2
124      pmulhw xmm4, xmm7        ; tm03*tan2
125      movdqa xmm5, [tan2]     ; xmm5 <= tan2
126      psubsw xmm4, xmm6        ; out6 = tm03*tan2 - tm12
127      pmulhw xmm5, xmm6        ; tm12*tan2
128      paddsw xmm5, xmm7        ; out2 = tm12*tan2 + tm03

130      movdqa xmm6, [sqrt2]
131      movdqa xmm7, [Rounder1]

133      pmulhw xmm2, xmm6        ; xmm2: tp65 = (t6 + t5)*cos4
134      por    xmm5, xmm7        ; correct out2
135      por    xmm4, xmm7        ; correct out6
136      pmulhw xmm1, xmm6        ; xmm1: tm65 = (t6 - t5)*cos4
137      por    xmm2, xmm7        ; correct tp65

139      movdqa [ecx+2*16], xmm5  ; => out2
140      movdqa xmm5, xmm3        ; save t4
141      movdqa [ecx+6*16], xmm4  ; => out6
142      movdqa xmm4, xmm0        ; save t7

144      psubsw xmm3, xmm1        ; xmm3: tm465 = t4 - tm65
145      psubsw xmm0, xmm2        ; xmm0: tm765 = t7 - tp65
146      paddsw xmm2, xmm4        ; xmm2: tp765 = t7 + tp65
147      paddsw xmm1, xmm5        ; xmm1: tp465 = t4 + tm65

149      movdqa xmm4, [tan3]     ; tan3 - 1
150      movdqa xmm5, [tan1]     ; tan1

152      movdqa xmm7, xmm3        ; save tm465
153      pmulhw xmm3, xmm4        ; tm465*(tan3-1)
```

```

154      movdqa xmm6, xmm1      ; save tp465
155      pmulhw xmm1, xmm5      ; tp465*tan1

157      paddsw xmm3, xmm7      ; tm465*tan3
158      pmulhw xmm4, xmm0      ; tm765*(tan3-1)
159      paddsw xmm4, xmm0      ; tm765*tan3
160      pmulhw xmm5, xmm2      ; tp765*tan1

162      paddsw xmm1, xmm2      ; out1 = tp765 + tp465*tan1
163      psubsw xmm0, xmm3      ; out3 = tm765 - tm465*tan3
164      paddsw xmm7, xmm4      ; out5 = tm465 + tm765*tan3
165      psubsw xmm5, xmm6      ; out7 =-tp465 + tp765*tan1

167      movdqa [ecx+1*16], xmm1 ; => out1
168      movdqa [ecx+3*16], xmm0 ; => out3
169      movdqa [ecx+5*16], xmm7 ; => out5
170      movdqa [ecx+7*16], xmm5 ; => out7
171  }

173  __asm // fMTX_MULT 0, fTab1, Fdct_Rnd0
174  {
175      movdqa  xmm0, [ecx+0]    ; xmm0 = [0123][4567]
176      pshufhw xmm1, xmm0, 00011011b ; xmm1 = [----][7654]
177      pshufd  xmm0, xmm0, 01000100b
178      pshufd  xmm1, xmm1, 11101110b

180      movdqa  xmm2, xmm0
181      paddsw  xmm0, xmm1      ; xmm0 = [a0 a1 a2 a3]
182      psubsw  xmm2, xmm1      ; xmm2 = [b0 b1 b2 b3]

184      punpckldq xmm0, xmm2    ; xmm0 = [a0 a1 b0 b1][a2 a3 b2 b3]
185      pshufd   xmm2, xmm0, 01001110b ; xmm2 = [a2 a3 b2 b3][a0 a1 b0 b1]

187      movdqa  xmm1, [fTab1+16]
188      movdqa  xmm3, [fTab1+32]
189      pmaddwd  xmm1, xmm2
190      pmaddwd  xmm3, xmm0
191      pmaddwd  xmm2, [fTab1+48]
192      pmaddwd  xmm0, [fTab1+ 0]

194      padddd  xmm0, xmm1      ; [ out0 | out1 ][ out2 | out3 ]
195      padddd  xmm2, xmm3      ; [ out4 | out5 ][ out6 | out7 ]
196      psrad   xmm0, 16
197      psrad   xmm2, 16

199      packssdw xmm0, xmm2      ; [ out0 .. out7 ]

```

```
200      paddsw    xmm0, [Fdct_Rnd0]          ; Round

202      psraw     xmm0, 4                    ; => [-2048, 2047]

204      movdqa    [ecx+0], xmm0
205  }

207  __asm //    fMTX_MULT  1, fTab2, Fdct_Rnd2
208  {
209      movdqa    xmm0, [ecx+16]      ; xmm0 = [0123][4567]
210      pshufhw   xmm1, xmm0, 00011011b ; xmm1 = [----][7654]
211      pshufd    xmm0, xmm0, 01000100b
212      pshufd    xmm1, xmm1, 11101110b

214      movdqa    xmm2, xmm0
215      paddsw    xmm0, xmm1          ; xmm0 = [a0 a1 a2 a3]
216      psubsw    xmm2, xmm1          ; xmm2 = [b0 b1 b2 b3]

218      punpckldq xmm0, xmm2          ; xmm0 = [a0 a1 b0 b1][a2 a3 b2 b3]
219      pshufd    xmm2, xmm0, 01001110b ; xmm2 = [a2 a3 b2 b3][a0 a1 b0 b1]

221      movdqa    xmm1, [fTab2+16]
222      movdqa    xmm3, [fTab2+32]
223      pmaddwd   xmm1, xmm2
224      pmaddwd   xmm3, xmm0
225      pmaddwd   xmm2, [fTab2+48]
226      pmaddwd   xmm0, [fTab2+ 0]

228      padd     xmm0, xmm1          ; [ out0 | out1 ][ out2 | out3 ]
229      padd     xmm2, xmm3          ; [ out4 | out5 ][ out6 | out7 ]
230      psrad    xmm0, 16
231      psrad    xmm2, 16

233      packssdw xmm0, xmm2          ; [ out0 .. out7 ]
234      paddsw    xmm0, [Fdct_Rnd2]      ; Round

236      psraw     xmm0, 4                    ; => [-2048, 2047]

238      movdqa    [ecx+16], xmm0
239  }

241  __asm //    fMTX_MULT  2, fTab3, Fdct_Rnd1
242  {
243      movdqa    xmm0, [ecx+32]      ; xmm0 = [0123][4567]
244      pshufhw   xmm1, xmm0, 00011011b ; xmm1 = [----][7654]
245      pshufd    xmm0, xmm0, 01000100b
```

```

246      pshufd    xmm1, xmm1, 11101110b

248      movdqa    xmm2, xmm0
249      paddsw    xmm0, xmm1                ; xmm0 = [a0 a1 a2 a3]
250      psubsw    xmm2, xmm1                ; xmm2 = [b0 b1 b2 b3]

252      punpckldq xmm0, xmm2                ; xmm0 = [a0 a1 b0 b1][a2 a3 b2 b3]
253      pshufd    xmm2, xmm0, 01001110b ; xmm2 = [a2 a3 b2 b3][a0 a1 b0 b1]

255      movdqa    xmm1, [fTab3+16]
256      movdqa    xmm3, [fTab3+32]
257      pmaddwd   xmm1, xmm2
258      pmaddwd   xmm3, xmm0
259      pmaddwd   xmm2, [fTab3+48]
260      pmaddwd   xmm0, [fTab3+ 0]

262      padddd    xmm0, xmm1                ; [ out0 | out1 ][ out2 | out3 ]
263      padddd    xmm2, xmm3                ; [ out4 | out5 ][ out6 | out7 ]
264      psrad     xmm0, 16
265      psrad     xmm2, 16

267      packssdw  xmm0, xmm2                ; [ out0 .. out7 ]
268      paddsw    xmm0, [Fdct_Rnd1]          ; Round

270      psraw     xmm0, 4                    ; => [-2048, 2047]

272      movdqa    [ecx+32], xmm0
273  }

275  __asm //  fMTX_MULT  3, fTab4, Fdct_Rnd1
276  {
277      movdqa    xmm0, [ecx+48]    ; xmm0 = [0123][4567]
278      pshufhw   xmm1, xmm0, 00011011b ; xmm1 = [----][7654]
279      pshufd    xmm0, xmm0, 01000100b
280      pshufd    xmm1, xmm1, 11101110b

282      movdqa    xmm2, xmm0
283      paddsw    xmm0, xmm1                ; xmm0 = [a0 a1 a2 a3]
284      psubsw    xmm2, xmm1                ; xmm2 = [b0 b1 b2 b3]

286      punpckldq xmm0, xmm2                ; xmm0 = [a0 a1 b0 b1][a2 a3 b2 b3]
287      pshufd    xmm2, xmm0, 01001110b ; xmm2 = [a2 a3 b2 b3][a0 a1 b0 b1]

289      movdqa    xmm1, [fTab4+16]
290      movdqa    xmm3, [fTab4+32]
291      pmaddwd   xmm1, xmm2

```



```
292      pmaddwd xmm3, xmm0
293      pmaddwd xmm2, [fTab4+48]
294      pmaddwd xmm0, [fTab4+ 0]

296      paddb   xmm0, xmm1          ; [ out0 | out1 ][ out2 | out3 ]
297      paddb   xmm2, xmm3          ; [ out4 | out5 ][ out6 | out7 ]
298      psrad   xmm0, 16
299      psrad   xmm2, 16

301      packssdw xmm0, xmm2          ; [ out0 .. out7 ]
302      paddsw   xmm0, [Fdct_Rnd1]    ; Round

304      psraw   xmm0, 4              ; => [-2048, 2047]

306      movdqa  [ecx+48], xmm0
307  }
308  __asm //  fMTX_MULT  4, fTab1, Fdct_Rnd0
309  {
310      movdqa  xmm0, [ecx+64]    ; xmm0 = [0123][4567]
311      pshufhw xmm1, xmm0, 00011011b ; xmm1 = [----][7654]
312      pshufd  xmm0, xmm0, 01000100b
313      pshufd  xmm1, xmm1, 11101110b

315      movdqa  xmm2, xmm0
316      paddsw  xmm0, xmm1          ; xmm0 = [a0 a1 a2 a3]
317      psubsw  xmm2, xmm1          ; xmm2 = [b0 b1 b2 b3]

319      punpckldq xmm0, xmm2          ; xmm0 = [a0 a1 b0 b1][a2 a3 b2 b3]
320      pshufd  xmm2, xmm0, 01001110b ; xmm2 = [a2 a3 b2 b3][a0 a1 b0 b1]

322      movdqa  xmm1, [fTab1+16]
323      movdqa  xmm3, [fTab1+32]
324      pmaddwd xmm1, xmm2
325      pmaddwd xmm3, xmm0
326      pmaddwd xmm2, [fTab1+48]
327      pmaddwd xmm0, [fTab1+ 0]

329      paddb   xmm0, xmm1          ; [ out0 | out1 ][ out2 | out3 ]
330      paddb   xmm2, xmm3          ; [ out4 | out5 ][ out6 | out7 ]
331      psrad   xmm0, 16
332      psrad   xmm2, 16

334      packssdw xmm0, xmm2          ; [ out0 .. out7 ]
335      paddsw   xmm0, [Fdct_Rnd0]    ; Round

337      psraw   xmm0, 4              ; => [-2048, 2047]
```

```
339     movdqa [ecx+64], xmm0
340 }
341 __asm //  fMTX_MULT  5, fTab4, Fdct_Rnd1
342 {
343     movdqa  xmm0, [ecx+80]    ; xmm0 = [0123][4567]
344     pshufhw xmm1, xmm0, 00011011b ; xmm1 = [----][7654]
345     pshufd  xmm0, xmm0, 01000100b
346     pshufd  xmm1, xmm1, 11101110b

348     movdqa  xmm2, xmm0
349     paddsw  xmm0, xmm1        ; xmm0 = [a0 a1 a2 a3]
350     psubsw  xmm2, xmm1        ; xmm2 = [b0 b1 b2 b3]

352     punpckldq xmm0, xmm2      ; xmm0 = [a0 a1 b0 b1][a2 a3 b2 b3]
353     pshufd  xmm2, xmm0, 01001110b ; xmm2 = [a2 a3 b2 b3][a0 a1 b0 b1]

355     movdqa  xmm1, [fTab4+16]
356     movdqa  xmm3, [fTab4+32]
357     pmaddwd xmm1, xmm2
358     pmaddwd xmm3, xmm0
359     pmaddwd xmm2, [fTab4+48]
360     pmaddwd xmm0, [fTab4+ 0]

362     paddd   xmm0, xmm1        ; [ out0 | out1 ][ out2 | out3 ]
363     paddd   xmm2, xmm3        ; [ out4 | out5 ][ out6 | out7 ]
364     psrad   xmm0, 16
365     psrad   xmm2, 16

367     packssdw xmm0, xmm2      ; [ out0 .. out7 ]
368     paddsw   xmm0, [Fdct_Rnd1] ; Round

370     psraw   xmm0, 4          ; => [-2048, 2047]

372     movdqa  [ecx+80], xmm0
373 }
374 __asm //  fMTX_MULT  6, fTab3, Fdct_Rnd1
375 {
376     movdqa  xmm0, [ecx+96]    ; xmm0 = [0123][4567]
377     pshufhw xmm1, xmm0, 00011011b ; xmm1 = [----][7654]
378     pshufd  xmm0, xmm0, 01000100b
379     pshufd  xmm1, xmm1, 11101110b

381     movdqa  xmm2, xmm0
382     paddsw  xmm0, xmm1        ; xmm0 = [a0 a1 a2 a3]
383     psubsw  xmm2, xmm1        ; xmm2 = [b0 b1 b2 b3]
```

```
385     punpckldq xmm0, xmm2           ; xmm0 = [a0 a1 b0 b1][a2 a3 b2 b3]
386     pshufd     xmm2, xmm0, 01001110b ; xmm2 = [a2 a3 b2 b3][a0 a1 b0 b1]

388     movdqa     xmm1, [fTab3+16]
389     movdqa     xmm3, [fTab3+32]
390     pmaddwd    xmm1, xmm2
391     pmaddwd    xmm3, xmm0
392     pmaddwd    xmm2, [fTab3+48]
393     pmaddwd    xmm0, [fTab3+ 0]

395     paddb     xmm0, xmm1           ; [ out0 | out1 ][ out2 | out3 ]
396     paddb     xmm2, xmm3           ; [ out4 | out5 ][ out6 | out7 ]
397     psrad     xmm0, 16
398     psrad     xmm2, 16

400     packssdw  xmm0, xmm2           ; [ out0 .. out7 ]
401     paddsw    xmm0, [Fdct_Rnd1]     ; Round

403     psraw     xmm0, 4              ; => [-2048, 2047]

405     movdqa    [ecx+96], xmm0
406 }
407 __asm //  fMTX_MULT  7, fTab2, Fdct_Rnd1
408 {
409     movdqa     xmm0, [ecx+112]      ; xmm0 = [0123][4567]
410     pshufhw    xmm1, xmm0, 00011011b ; xmm1 = [----][7654]
411     pshufd     xmm0, xmm0, 01000100b
412     pshufd     xmm1, xmm1, 11101110b

414     movdqa     xmm2, xmm0
415     paddsw     xmm0, xmm1           ; xmm0 = [a0 a1 a2 a3]
416     psubsw     xmm2, xmm1           ; xmm2 = [b0 b1 b2 b3]

418     punpckldq xmm0, xmm2           ; xmm0 = [a0 a1 b0 b1][a2 a3 b2 b3]
419     pshufd     xmm2, xmm0, 01001110b ; xmm2 = [a2 a3 b2 b3][a0 a1 b0 b1]

421     movdqa     xmm1, [fTab2+16]
422     movdqa     xmm3, [fTab2+32]
423     pmaddwd    xmm1, xmm2
424     pmaddwd    xmm3, xmm0
425     pmaddwd    xmm2, [fTab2+48]
426     pmaddwd    xmm0, [fTab2+ 0]

428     paddb     xmm0, xmm1           ; [ out0 | out1 ][ out2 | out3 ]
429     paddb     xmm2, xmm3           ; [ out4 | out5 ][ out6 | out7 ]
```

```
430         psrad    xmm0, 16
431         psrad    xmm2, 16

433         packssdw xmm0, xmm2           ; [ out0 .. out7 ]
434         paddsw   xmm0, [Fdct_Rnd1]    ; Round

436         psraw   xmm0, 4               ; => [-2048, 2047]

438         movdqa  [ecx+112], xmm0
439     }
440 }

443 #else

446 #define USE_ACCURATE_ROUNDING

448 #define RIGHT_SHIFT(x, shft)  ((x) >> (shft))

450 #ifdef USE_ACCURATE_ROUNDING
451 #define ONE ((int) 1)
452 #define DESCALE(x, n)  RIGHT_SHIFT((x) + (ONE << ((n) - 1)), n)
453 #else
454 #define DESCALE(x, n)  RIGHT_SHIFT(x, n)
455 #endif

457 #define CONST_BITS  13
458 #define PASS1_BITS  2

460 #define FIX_0_298631336 ((int) 2446) /* FIX(0.298631336) */
461 #define FIX_0_390180644 ((int) 3196) /* FIX(0.390180644) */
462 #define FIX_0_541196100 ((int) 4433) /* FIX(0.541196100) */
463 #define FIX_0_765366865 ((int) 6270) /* FIX(0.765366865) */
464 #define FIX_0_899976223 ((int) 7373) /* FIX(0.899976223) */
465 #define FIX_1_175875602 ((int) 9633) /* FIX(1.175875602) */
466 #define FIX_1_501321110 ((int) 12299) /* FIX(1.501321110) */
467 #define FIX_1_847759065 ((int) 15137) /* FIX(1.847759065) */
468 #define FIX_1_961570560 ((int) 16069) /* FIX(1.961570560) */
469 #define FIX_2_053119869 ((int) 16819) /* FIX(2.053119869) */
470 #define FIX_2_562915447 ((int) 20995) /* FIX(2.562915447) */
471 #define FIX_3_072711026 ((int) 25172) /* FIX(3.072711026) */

473 void fdct(short *const block)
474 {
475     int tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
```

```
476     int tmp10, tmp11, tmp12, tmp13;
477     int z1, z2, z3, z4, z5;
478     short *blkptr;
479     int *dataptr;
480     int data[64];
481     int i;

483     /* Pass 1: process rows. */
484     /* Note results are scaled up by sqrt(8) compared to a true DCT; */
485     /* furthermore, we scale the results by 2**PASS1_BITS. */

487     dataptr = data;
488     blkptr = block;
489     for (i = 0; i < 8; i++)
490     {
491         tmp0 = blkptr[0] + blkptr[7];
492         tmp7 = blkptr[0] - blkptr[7];
493         tmp1 = blkptr[1] + blkptr[6];
494         tmp6 = blkptr[1] - blkptr[6];
495         tmp2 = blkptr[2] + blkptr[5];
496         tmp5 = blkptr[2] - blkptr[5];
497         tmp3 = blkptr[3] + blkptr[4];
498         tmp4 = blkptr[3] - blkptr[4];

500         /* Even part per LL&M figure 1 --- note that published figure is faulty;
501          * rotator "sqrt(2)*c1" should be "sqrt(2)*c6".
502          */

504         tmp10 = tmp0 + tmp3;
505         tmp13 = tmp0 - tmp3;
506         tmp11 = tmp1 + tmp2;
507         tmp12 = tmp1 - tmp2;

509         dataptr[0] = (tmp10 + tmp11) << PASS1_BITS;
510         dataptr[4] = (tmp10 - tmp11) << PASS1_BITS;

512         z1 = (tmp12 + tmp13) * FIX_0_541196100;
513         dataptr[2] =
514             DESCALE(z1 + tmp13 * FIX_0_765366865, CONST_BITS - PASS1_BITS);
515         dataptr[6] =
516             DESCALE(z1 + tmp12 * (-FIX_1_847759065), CONST_BITS - PASS1_BITS);

518         /* Odd part per figure 8 --- note paper omits factor of sqrt(2).
519          * cK represents cos(K*pi/16).
520          * i0..i3 in the paper are tmp4..tmp7 here.
521          */
```

```
523     z1 = tmp4 + tmp7;
524     z2 = tmp5 + tmp6;
525     z3 = tmp4 + tmp6;
526     z4 = tmp5 + tmp7;
527     z5 = (z3 + z4) * FIX_1_175875602;    /* sqrt(2) * c3 */

529     tmp4 *= FIX_0_298631336;    /* sqrt(2) * (-c1+c3+c5-c7) */
530     tmp5 *= FIX_2_053119869;    /* sqrt(2) * ( c1+c3-c5+c7) */
531     tmp6 *= FIX_3_072711026;    /* sqrt(2) * ( c1+c3+c5-c7) */
532     tmp7 *= FIX_1_501321110;    /* sqrt(2) * ( c1+c3-c5-c7) */
533     z1 *= -FIX_0_899976223; /* sqrt(2) * (c7-c3) */
534     z2 *= -FIX_2_562915447; /* sqrt(2) * (-c1-c3) */
535     z3 *= -FIX_1_961570560; /* sqrt(2) * (-c3-c5) */
536     z4 *= -FIX_0_390180644; /* sqrt(2) * (c5-c3) */

538     z3 += z5;
539     z4 += z5;

541     dataptr[7] = DESCALE(tmp4 + z1 + z3, CONST_BITS - PASS1_BITS);
542     dataptr[5] = DESCALE(tmp5 + z2 + z4, CONST_BITS - PASS1_BITS);
543     dataptr[3] = DESCALE(tmp6 + z2 + z3, CONST_BITS - PASS1_BITS);
544     dataptr[1] = DESCALE(tmp7 + z1 + z4, CONST_BITS - PASS1_BITS);

546     dataptr += 8;    /* advance pointer to next row */
547     blkptr += 8;
548 }

550 /* Pass 2: process columns.
551  * We remove the PASS1_BITS scaling, but leave the results scaled up
552  * by an overall factor of 8.
553  */

555 dataptr = data;
556 for (i = 0; i < 8; i++)
557 {
558     tmp0 = dataptr[0] + dataptr[56];
559     tmp7 = dataptr[0] - dataptr[56];
560     tmp1 = dataptr[8] + dataptr[48];
561     tmp6 = dataptr[8] - dataptr[48];
562     tmp2 = dataptr[16] + dataptr[40];
563     tmp5 = dataptr[16] - dataptr[40];
564     tmp3 = dataptr[24] + dataptr[32];
565     tmp4 = dataptr[24] - dataptr[32];

567     /* Even part per LL&M figure 1 --- note that published figure is faulty;
```

```
568      * rotator "sqrt(2)*c1" should be "sqrt(2)*c6".
569      */

571      tmp10 = tmp0 + tmp3;
572      tmp13 = tmp0 - tmp3;
573      tmp11 = tmp1 + tmp2;
574      tmp12 = tmp1 - tmp2;

576      dataptr[0] = DESCALE(tmp10 + tmp11, PASS1_BITS);
577      dataptr[32] = DESCALE(tmp10 - tmp11, PASS1_BITS);

579      z1 = (tmp12 + tmp13) * FIX_0_541196100;
580      dataptr[16] =
581          DESCALE(z1 + tmp13 * FIX_0_765366865, CONST_BITS + PASS1_BITS);
582      dataptr[48] =
583          DESCALE(z1 + tmp12 * (-FIX_1_847759065), CONST_BITS + PASS1_BITS);

585      /* Odd part per figure 8 --- note paper omits factor of sqrt(2).
586      * cK represents cos(K*pi/16).
587      * i0..i3 in the paper are tmp4..tmp7 here.
588      */

590      z1 = tmp4 + tmp7;
591      z2 = tmp5 + tmp6;
592      z3 = tmp4 + tmp6;
593      z4 = tmp5 + tmp7;
594      z5 = (z3 + z4) * FIX_1_175875602; /* sqrt(2) * c3 */

596      tmp4 *= FIX_0_298631336; /* sqrt(2) * (-c1+c3+c5-c7) */
597      tmp5 *= FIX_2_053119869; /* sqrt(2) * ( c1+c3-c5+c7) */
598      tmp6 *= FIX_3_072711026; /* sqrt(2) * ( c1+c3+c5-c7) */
599      tmp7 *= FIX_1_501321110; /* sqrt(2) * ( c1+c3-c5-c7) */
600      z1 *= -FIX_0_899976223; /* sqrt(2) * (c7-c3) */
601      z2 *= -FIX_2_562915447; /* sqrt(2) * (-c1-c3) */
602      z3 *= -FIX_1_961570560; /* sqrt(2) * (-c3-c5) */
603      z4 *= -FIX_0_390180644; /* sqrt(2) * (c5-c3) */

605      z3 += z5;
606      z4 += z5;

608      dataptr[56] = DESCALE(tmp4 + z1 + z3, CONST_BITS + PASS1_BITS);
609      dataptr[40] = DESCALE(tmp5 + z2 + z4, CONST_BITS + PASS1_BITS);
610      dataptr[24] = DESCALE(tmp6 + z2 + z3, CONST_BITS + PASS1_BITS);
611      dataptr[8] = DESCALE(tmp7 + z1 + z4, CONST_BITS + PASS1_BITS);

613      dataptr++; /* advance pointer to next column */
```

```
614     }
615     /* descale */
616     for (i = 0; i < 64; i++)
617         block[i] = (short int) DESCALE(data[i], 3);
618 }

620 #endif
```

6.4 idct.h 文件

6.4.1 功能描述

反向 8x8 二维 DCT 变换函数原型的声明。

6.4.2 文件注释

```
1  #ifndef _IDCT_H_
2  #define _IDCT_H_

4  void idct_int32_init();

6  void idct(short *const block);

8  #endif
```

6.5 idct.c 文件

6.5.1 功能描述

反向 8x8 二维 DCT 变换函数的实现，请查阅其他相关书籍，包括 C 语言和 PC 汇编语言版本。

6.5.2 文件注释

```
1  #include "../portab.h"
2  #include "idct.h"

4  #ifdef IDCT_ACC

6  #define BITS_INV_ACC    4           // 4 or 5 for IEEE
7  #define SHIFT_INV_ROW  (16 - BITS_INV_ACC)
8  #define SHIFT_INV_COL  (1 + BITS_INV_ACC)
9  #define RND_INV_ROW    (1024 * (6 - BITS_INV_ACC)) //1 << (SHIFT_INV_ROW-1)
10 #define RND_INV_COL    (16 * (BITS_INV_ACC - 3))   //1 << (SHIFT_INV_COL-1)
11 #define RND_INV_CORR   (RND_INV_COL - 1)          // correction -1.0 and round
```



```
13 __declspec(aligned(16)) short M128_one_corr[8] = {1, 1, 1, 1, 1, 1, 1, 1};
14 __declspec(aligned(16)) short M128_round_inv_row[8] = {
15     RND_INV_ROW, 0, RND_INV_ROW, 0, RND_INV_ROW, 0, RND_INV_ROW, 0};
16 //Is16vec8 tomorrow = *(Is16vec8*)M128_round_inv_row;

18 __declspec(aligned(16)) short M128_round_inv_col[8] = {
19     RND_INV_COL, RND_INV_COL, RND_INV_COL, RND_INV_COL,
20     RND_INV_COL, RND_INV_COL, RND_INV_COL, RND_INV_COL};

22 __declspec(aligned(16)) short M128_round_inv_corr[8]= {
23     RND_INV_CORR, RND_INV_CORR, RND_INV_CORR, RND_INV_CORR,
24     RND_INV_CORR, RND_INV_CORR, RND_INV_CORR, RND_INV_CORR};
25 //round_frw_row dword RND_FRW_ROW, RND_FRW_ROW

27 __declspec(aligned(16)) short M128_tg_1_16[8] = {
28     13036, 13036, 13036, 13036,
29     13036, 13036, 13036, 13036}; // tg * (2<<16) + 0.5

31 __declspec(aligned(16)) short M128_tg_2_16[8] = {
32     27146, 27146, 27146, 27146,
33     27146, 27146, 27146, 27146}; // tg * (2<<16) + 0.5

35 __declspec(aligned(16)) short M128_tg_3_16[8] = {
36     -21746, -21746, -21746, -21746,
37     -21746, -21746, -21746, -21746}; // tg * (2<<16) + 0.5

39 __declspec(aligned(16)) short M128_cos_4_16[8] = {
40     -19195, -19195, -19195, -19195,
41     -19195, -19195, -19195, -19195}; // cos * (2<<16) + 0.5

43 //ocos_4_16 sword 23170, 23170, 23170, 23170 ; cos * (2<<15) + 0.5

45 __declspec(aligned(16)) short M128_tab_i_04[] = {
46     16384, 21407, 16384, 8867, //movq -> w05 w04 w01 w00
47     16384, -8867, 16384, -21407, // w13 w12 w09 w08
48     16384, 8867, -16384, -21407, // w07 w06 w03 w02
49     -16384, 21407, 16384, -8867, // w15 w14 w11 w10
50     22725, 19266, 19266, -4520, // w21 w20 w17 w16
51     12873, -22725, 4520, -12873, // w29 w28 w25 w24
52     12873, 4520, -22725, -12873, // w23 w22 w19 w18
```

```
53      4520,   19266,  19266,  -22725};    //          w31 w30 w27 w26

55  __declspec(aligned(16)) short M128_tab_i_17[] = {
56      22725,  29692,  22725,  12299,      //movq ->  w05 w04 w01 w00
57      22725, -12299, 22725,  -29692,      //          w13 w12 w09 w08
58      22725,  12299, -22725, -29692,      //          w07 w06 w03 w02
59      -22725, 29692,  22725, -12299,      //          w15 w14 w11 w10
60      31521,  26722,  26722,  -6270,      //          w21 w20 w17 w16
61      17855, -31521, 6270,   -17855,      //          w29 w28 w25 w24
62      17855,  6270,   -31521, -17855,      //          w23 w22 w19 w18
63      6270,   26722,  26722,  -31521};    //          w31 w30 w27 w26

65  __declspec(aligned(16)) short M128_tab_i_26[] = {
66      21407,  27969,  21407,  11585,      //movq ->  w05 w04 w01 w00
67      21407, -11585, 21407,  -27969,      //          w13 w12 w09 w08
68      21407,  11585, -21407, -27969,      //          w07 w06 w03 w02
69      -21407, 27969,  21407, -11585,      //          w15 w14 w11 w10
70      29692,  25172,  25172,  -5906,      //          w21 w20 w17 w16
71      16819, -29692, 5906,   -16819,      //          w29 w28 w25 w24
72      16819,  5906,   -29692, -16819,      //          w23 w22 w19 w18
73      5906,   25172,  25172,  -29692};    //          w31 w30 w27 w26

75  __declspec(aligned(16)) short M128_tab_i_35[] = {
76      19266,  25172,  19266,  10426,      //movq ->  w05 w04 w01 w00
77      19266, -10426, 19266,  -25172,      //          w13 w12 w09 w08
78      19266,  10426, -19266, -25172,      //          w07 w06 w03 w02
79      -19266, 25172,  19266, -10426,      //          w15 w14 w11 w10
80      26722,  22654,  22654,  -5315,      //          w21 w20 w17 w16
81      15137, -26722, 5315,   -15137,      //          w29 w28 w25 w24
82      15137,  5315,   -26722, -15137,      //          w23 w22 w19 w18
83      5315,   22654,  22654,  -26722};    //          w31 w30 w27 w26

85  //-----

87  //xmm7 = round_inv_row

89  #define DCT_8_INV_ROW    __asm{          \
90      __asm    pshufw      xmm0, xmm0, 0xD8  \
91      __asm    pshufhw     xmm0, xmm0, 0xD8  \
92      __asm    pshufd      xmm3, xmm0, 0x55  \
```

```
93     __asm    pshufd    xmm1, xmm0, 0           \
94     __asm    pshufd    xmm2, xmm0, 0xAA        \
95     __asm    pshufd    xmm0, xmm0, 0xFF        \
96     __asm    pmaddwd    xmm1, [esi]            \
97     __asm    pmaddwd    xmm2, [esi+16]         \
98     __asm    pmaddwd    xmm3, [esi+32]         \
99     __asm    pmaddwd    xmm0, [esi+48]         \
100    __asm    paddb      xmm0, xmm3             \
101    __asm    pshufb     xmm4, xmm4, 0xD8        \
102    __asm    pshufb     xmm4, xmm4, 0xD8        \
103    __asm    movdqa     xmm7, M128_round_inv_row \
104    __asm    paddb      xmm1, xmm7             \
105    __asm    pshufd     xmm6, xmm4, 0xAA        \
106    __asm    pshufd     xmm5, xmm4, 0          \
107    __asm    pmaddwd    xmm5, [ecx]            \
108    __asm    paddb      xmm5, xmm7             \
109    __asm    pmaddwd    xmm6, [ecx+16]         \
110    __asm    pshufd     xmm7, xmm4, 0x55        \
111    __asm    pmaddwd    xmm7, [ecx+32]         \
112    __asm    pshufd     xmm4, xmm4, 0xFF        \
113    __asm    pmaddwd    xmm4, [ecx+48]         \
114    __asm    paddb      xmm1, xmm2             \
115    __asm    movdqa     xmm2, xmm1             \
116    __asm    psubb      xmm2, xmm0             \
117    __asm    psrad      xmm2, 12               \
118    __asm    pshufd     xmm2, xmm2, 0x1B        \
119    __asm    paddb      xmm0, xmm1             \
120    __asm    psrad      xmm0, 12               \
121    __asm    paddb      xmm5, xmm6             \
122    __asm    packssdw   xmm0, xmm2             \
123    __asm    paddb      xmm4, xmm7             \
124    __asm    movdqa     xmm6, xmm5             \
125    __asm    psubb      xmm6, xmm4             \
126    __asm    psrad      xmm6, 12               \
127    __asm    paddb      xmm4, xmm5             \
128    __asm    psrad      xmm4, 12               \
129    __asm    pshufd     xmm6, xmm6, 0x1B        \
130    __asm    packssdw   xmm4, xmm6             \
131 }
```

```
134 #define DCT_8_INV_COL_8 __asm{ \
135     __asm    movdqa    xmm6, xmm4 \
136     __asm    movdqa    xmm2, xmm0 \
137     __asm    movdqa    xmm3, XMMWORD PTR [edx+3*16] \
138     __asm    movdqa    xmm1, XMMWORD PTR M128_tg_3_16 \
139     __asm    pmulhw     xmm0, xmm1 \
140     __asm    movdqa    xmm5, XMMWORD PTR M128_tg_1_16 \
141     __asm    pmulhw     xmm1, xmm3 \
142     __asm    paddsw     xmm1, xmm3 \
143     __asm    pmulhw     xmm4, xmm5 \
144     __asm    movdqa    xmm7, XMMWORD PTR [edx+6*16] \
145     __asm    pmulhw     xmm5, [edx+1*16] \
146     __asm    psubsw     xmm5, xmm6 \
147     __asm    movdqa    xmm6, xmm5 \
148     __asm    paddsw     xmm4, [edx+1*16] \
149     __asm    paddsw     xmm0, xmm2 \
150     __asm    paddsw     xmm0, xmm3 \
151     __asm    psubsw     xmm2, xmm1 \
152     __asm    movdqa    xmm1, xmm0 \
153     __asm    movdqa    xmm3, XMMWORD PTR M128_tg_2_16 \
154     __asm    pmulhw     xmm7, xmm3 \
155     __asm    pmulhw     xmm3, [edx+2*16] \
156     __asm    paddsw     xmm0, xmm4 \
157     __asm    psubsw     xmm4, xmm1 \
158     __asm    paddsw     xmm0, XMMWORD PTR M128_one_corr \
159     __asm    movdqa    [edx+7*16], xmm0 \
160     __asm    psubsw     xmm5, xmm2 \
161     __asm    paddsw     xmm5, XMMWORD PTR M128_one_corr \
162     __asm    paddsw     xmm6, xmm2 \
163     __asm    movdqa    [edx+3*16], xmm6 \
164     __asm    movdqa    xmm1, xmm4 \
165     __asm    movdqa    xmm0, XMMWORD PTR M128_cos_4_16 \
166     __asm    movdqa    xmm2, xmm0 \
167     __asm    paddsw     xmm4, xmm5 \
168     __asm    psubsw     xmm1, xmm5 \
169     __asm    paddsw     xmm7, [edx+2*16] \
170     __asm    psubsw     xmm3, [edx+6*16] \
171     __asm    movdqa    xmm6, [edx] \
172     __asm    pmulhw     xmm0, xmm1 \
```

173	__asm	movdqa	xmm5, [edx+4*16]	\
174	__asm	paddsw	xmm5, xmm6	\
175	__asm	psubsw	xmm6, [edx+4*16]	\
176	__asm	pmulhw	xmm2, xmm4	\
177	__asm	paddsw	xmm4, xmm2	\
178	__asm	movdqa	xmm2, xmm5	\
179	__asm	psubsw	xmm2, xmm7	\
180	__asm	por	xmm4, XMMWORD PTR M128_one_corr	\
181	__asm	paddsw	xmm0, xmm1	\
182	__asm	por	xmm0, XMMWORD PTR M128_one_corr	\
183	__asm	paddsw	xmm5, xmm7	\
184	__asm	paddsw	xmm5, XMMWORD PTR M128_round_inv_col	\
185	__asm	movdqa	xmm1, xmm6	\
186	__asm	movdqa	xmm7, [edx+7*16]	\
187	__asm	paddsw	xmm7, xmm5	\
188	__asm	psraw	xmm7, SHIFT_INV_COL	\
189	__asm	movdqa	[edx], xmm7	\
190	__asm	paddsw	xmm6, xmm3	\
191	__asm	paddsw	xmm6, XMMWORD PTR M128_round_inv_col	\
192	__asm	psubsw	xmm1, xmm3	\
193	__asm	paddsw	xmm1, XMMWORD PTR M128_round_inv_corr	\
194	__asm	movdqa	xmm7, xmm1	\
195	__asm	movdqa	xmm3, xmm6	\
196	__asm	paddsw	xmm6, xmm4	\
197	__asm	paddsw	xmm2, XMMWORD PTR M128_round_inv_corr	\
198	__asm	psraw	xmm6, SHIFT_INV_COL	\
199	__asm	movdqa	[edx+1*16], xmm6	\
200	__asm	paddsw	xmm1, xmm0	\
201	__asm	psraw	xmm1, SHIFT_INV_COL	\
202	__asm	movdqa	[edx+2*16], xmm1	\
203	__asm	movdqa	xmm1, [edx+3*16]	\
204	__asm	movdqa	xmm6, xmm1	\
205	__asm	psubsw	xmm7, xmm0	\
206	__asm	psraw	xmm7, SHIFT_INV_COL	\
207	__asm	movdqa	[edx+5*16], xmm7	\
208	__asm	psubsw	xmm5, [edx+7*16]	\
209	__asm	psraw	xmm5, SHIFT_INV_COL	\
210	__asm	movdqa	[edx+7*16], xmm5	\
211	__asm	psubsw	xmm3, xmm4	\
212	__asm	paddsw	xmm6, xmm2	\

```

213     __asm    psubsw    xmm2, xmm1                \
214     __asm    psraw    xmm6, SHIFT_INV_COL        \
215     __asm    movdqa    [edx+3*16],  xmm6          \
216     __asm    psraw    xmm2, SHIFT_INV_COL        \
217     __asm    movdqa    [edx+4*16],  xmm2          \
218     __asm    psraw    xmm3, SHIFT_INV_COL        \
219     __asm    movdqa    [edx+6*16],  xmm3          \
220 }

222 void idct_int32_init()
223 {
224 }

226 void idct(short *const block) //assumes block is aligned on a 16-byte boundary!
227 {
228     //MM_ALIGN16 short dst_block[8][8];
229     short* src = block;
230     short* dst = block; //dst_block[0];
231     // assert(((src & 0xf) == 0) && ((dst & 0xf) == 0))

233     __asm    mov        eax, src
234     __asm    movdqa     xmm0, XMMWORD PTR[eax] //row 1
235     __asm    movdqa     xmm4, XMMWORD PTR[eax+16*2] //row 3
236     __asm    mov        edx, dst
237     //.....//
238     __asm    lea        esi, M128_tab_i_04
239     __asm    lea        ecx, M128_tab_i_26
240     DCT_8_INV_ROW; //Row 1, tab_i_04 and Row 3, tab_i_26
241     __asm    movdqa     XMMWORD PTR[edx],      xmm0
242     __asm    movdqa     XMMWORD PTR[edx+16*2],  xmm4
243     //.....//
244     __asm    movdqa     xmm0, XMMWORD PTR[eax+16*4] //row 5
245     //__asm lea        esi, M128_tab_i_04
246     __asm    movdqa     xmm4, XMMWORD PTR[eax+16*6] //row 7
247     //__asm lea        ecx, M128_tab_i_26
248     DCT_8_INV_ROW; //Row 5, tab_i_04 and Row 7, tab_i_26
249     __asm    movdqa     XMMWORD PTR[edx+16*4],  xmm0
250     __asm    movdqa     XMMWORD PTR[edx+16*6],  xmm4
251     //.....//
252     __asm    movdqa     xmm0, XMMWORD PTR[eax+16*3] //row 4

```

```

253     __asm    lea        esi, M128_tab_i_35
254     __asm    movdqa     xmm4, XMMWORD PTR[eax+16*1] //row 2
255     __asm    lea        ecx, M128_tab_i_17
256     DCT_8_INV_ROW; //Row 4, tab_i_35 and Row 2, tab_i_17
257     __asm    movdqa     XMMWORD PTR[edx+16*3],  xmm0
258     __asm    movdqa     xmm0, XMMWORD PTR[eax+16*5] //row 6
259     __asm    movdqa     XMMWORD PTR[edx+16*1],  xmm4
260     //.....//
261     //__asm lea        esi, M128_tab_i_35
262     __asm    movdqa     xmm4, XMMWORD PTR[eax+16*7] //row 8
263     //__asm lea        ecx, M128_tab_i_17
264     DCT_8_INV_ROW; //Row 6, tab_i_35 and Row 8, tab_i_17
265     //__asm movdqa     XMMWORD PTR[edx+80],      xmm0
266     //__asm movdqa     xmm0, XMMWORD PTR [edx+80] /* 0          /* x5 */
267     //__asm movdqa     XMMWORD PTR[edx+16*7],  xmm4
268     //__asm movdqa     xmm4, XMMWORD PTR [edx+7*16]/* 4          ; x7 */

270     DCT_8_INV_COL_8
271 }

273 #else

275 #define W1 2841          /* 2048*sqrt(2)*cos(1*pi/16) */
276 #define W2 2676          /* 2048*sqrt(2)*cos(2*pi/16) */
277 #define W3 2408          /* 2048*sqrt(2)*cos(3*pi/16) */
278 #define W5 1609          /* 2048*sqrt(2)*cos(5*pi/16) */
279 #define W6 1108          /* 2048*sqrt(2)*cos(6*pi/16) */
280 #define W7 565           /* 2048*sqrt(2)*cos(7*pi/16) */

282 static short iclip[1024]; /* clipping table */
283 static short *icl;

285 void idct_int32_init()
286 {
287     int i;

289     icl = iclip + 512;
290     for (i = -512; i < 512; i++)
291         icl[i] = (i < -256) ? -256 : ((i > 255) ? 255 : i);
292 }

```

```
294 void idct(short *const block)
295 {
296     short *blk;
297     long i;
298     long X0, X1, X2, X3, X4, X5, X6, X7, X8;

301     for (i = 0; i < 8; i++)      /* idct rows */
302     {
303         blk = block + (i << 3);
304         if (!
305             ((X1 = blk[4] << 11) | (X2 = blk[6]) | (X3 = blk[2]) | (X4 = blk[1]) |
306              (X5 = blk[7]) | (X6 = blk[5]) | (X7 = blk[3])))
307         {
308             blk[0]=blk[1]=blk[2]=blk[3]=blk[4]=blk[5]=blk[6]=blk[7]=blk[0]<<3;
309             continue;
310         }

312         X0 = (blk[0] << 11) + 128; /* for proper rounding in the fourth stage */

314         /* first stage */
315         X8 = W7 * (X4 + X5);
316         X4 = X8 + (W1 - W7) * X4;
317         X5 = X8 - (W1 + W7) * X5;
318         X8 = W3 * (X6 + X7);
319         X6 = X8 - (W3 - W5) * X6;
320         X7 = X8 - (W3 + W5) * X7;

322         /* second stage */
323         X8 = X0 + X1;
324         X0 -= X1;
325         X1 = W6 * (X3 + X2);
326         X2 = X1 - (W2 + W6) * X2;
327         X3 = X1 + (W2 - W6) * X3;
328         X1 = X4 + X6;
329         X4 -= X6;
330         X6 = X5 + X7;
331         X5 -= X7;
```



```
333      /* third stage */
334      X7 = X8 + X3;
335      X8 -= X3;
336      X3 = X0 + X2;
337      X0 -= X2;
338      X2 = (181 * (X4 + X5) + 128) >> 8;
339      X4 = (181 * (X4 - X5) + 128) >> 8;

341      /* fourth stage */

343      blk[0] = (short) ((X7 + X1) >> 8);
344      blk[1] = (short) ((X3 + X2) >> 8);
345      blk[2] = (short) ((X0 + X4) >> 8);
346      blk[3] = (short) ((X8 + X6) >> 8);
347      blk[4] = (short) ((X8 - X6) >> 8);
348      blk[5] = (short) ((X0 - X4) >> 8);
349      blk[6] = (short) ((X3 - X2) >> 8);
350      blk[7] = (short) ((X7 - X1) >> 8);
351  } /* end for ( i = 0; i < 8; ++i ) IDCT-rows */

353  for (i = 0; i < 8; i++) /* idct columns */
354  {
355      blk = block + i;
356      /* shortcut */
357      if (!
358          ((X1 = (blk[32] << 8)) | (X2 = blk[48]) | (X3 = blk[16]) | (X4 = blk[8])
359           | (X5 = blk[56]) | (X6 = blk[40]) | (X7 = blk[24])))
360      {
361          blk[0]=blk[8]=blk[16]=blk[24]=blk[32]=blk[40]=blk[48]=blk[56]=
362              iclp[(blk[8 * 0] + 32) >> 6];

364          continue;
365      }

367      X0 = (blk[8 * 0] << 8) + 8192;

369      /* first stage */
370      X8 = W7 * (X4 + X5) + 4;
371      X4 = (X8 + (W1 - W7) * X4) >> 3;
372      X5 = (X8 - (W1 + W7) * X5) >> 3;
```

```
373      X8 = W3 * (X6 + X7) + 4;
374      X6 = (X8 - (W3 - W5) * X6) >> 3;
375      X7 = (X8 - (W3 + W5) * X7) >> 3;

377      /* second stage */
378      X8 = X0 + X1;
379      X0 -= X1;
380      X1 = W6 * (X3 + X2) + 4;
381      X2 = (X1 - (W2 + W6) * X2) >> 3;
382      X3 = (X1 + (W2 - W6) * X3) >> 3;
383      X1 = X4 + X6;
384      X4 -= X6;
385      X6 = X5 + X7;
386      X5 -= X7;

388      /* third stage */
389      X7 = X8 + X3;
390      X8 -= X3;
391      X3 = X0 + X2;
392      X0 -= X2;
393      X2 = (181 * (X4 + X5) + 128) >> 8;
394      X4 = (181 * (X4 - X5) + 128) >> 8;

396      /* fourth stage */
397      blk[8 * 0] = iclp[(X7 + X1) >> 14];
398      blk[8 * 1] = iclp[(X3 + X2) >> 14];
399      blk[8 * 2] = iclp[(X0 + X4) >> 14];
400      blk[8 * 3] = iclp[(X8 + X6) >> 14];
401      blk[8 * 4] = iclp[(X8 - X6) >> 14];
402      blk[8 * 5] = iclp[(X0 - X4) >> 14];
403      blk[8 * 6] = iclp[(X3 - X2) >> 14];
404      blk[8 * 7] = iclp[(X7 - X1) >> 14];
405      }
406 }

408 #endif
```

第七章 量化

7.1 文件列表

类型	名称	大小
	quant.h	472 bytes
	quant_h263.c	11410 bytes

7.2 quant.h 文件

7.2.1 功能描述

H263 方法量化和反量化函数原型的声明。

7.2.2 文件注释

```
1  #ifndef _QUANT_H_
2  #define _QUANT_H_

4  uint32_t quant_h263_intra(int16_t* coeff, const int16_t* data,
5                          const uint32_t quant, const uint32_t dscalar);
6  uint32_t dequant_h263_intra(int16_t* coeff, const int16_t* data,
7                          const uint32_t quant, const uint32_t dscalar);

9  uint32_t quant_h263_inter(int16_t* coeff, const int16_t* data,
10                         const uint32_t quant);
11 uint32_t dequant_h263_inter(int16_t* coeff, const int16_t* data,
12                         const uint32_t quant);

15 #endif
```

7.3 quant_h263.c 文件

7.3.1 功能描述

H263 方法量化和反量化函数的实现，包括 C 语言和 PC 汇编语言版本。

7.3.2 文件注释

```
1  #include "../portab.h"
2  #include "../global.h"

4  #include "quant.h"

6  #define SCALEBITS 16
```

```
7  #define FIX(X)      ((1L << SCALEBITS) / (2*X) + 1)

9  #ifdef QUANT_ACC

11  __declspec(aligned(16)) short plus_one[8]={1, 1, 1, 1, 1, 1, 1, 1};

13  __declspec(aligned(16)) short mmx_quant[]={
14      0,  0,  0,  0,
15      1,  1,  1,  1,
16      2,  2,  2,  2,
17      3,  3,  3,  3,
18      4,  4,  4,  4,
19      5,  5,  5,  5,
20      6,  6,  6,  6,
21      7,  7,  7,  7,
22      8,  8,  8,  8,
23      9,  9,  9,  9,
24      10, 10, 10, 10,
25      11, 11, 11, 11,
26      12, 12, 12, 12,
27      13, 13, 13, 13,
28      14, 14, 14, 14,
29      15, 15, 15, 15,
30      16, 16, 16, 16,
31      17, 17, 17, 17,
32      18, 18, 18, 18,
33      19, 19, 19, 19,
34      20, 20, 20, 20,
35      21, 21, 21, 21,
36      22, 22, 22, 22,
37      23, 23, 23, 23,
38      24, 24, 24, 24,
39      25, 25, 25, 25,
40      26, 26, 26, 26,
41      27, 27, 27, 27,
42      28, 28, 28, 28,
43      29, 29, 29, 29,
44      30, 30, 30, 30,
45      31, 31, 31, 31,
46      32, 32, 32, 32};

48  __declspec(aligned(16)) short mmx_sub[]={
49      0,  0,  0,  0,
50      1,  1,  1,  1,  1,  1,  1,  1,
51      2,  2,  2,  2,  2,  2,  2,  2,
52      3,  3,  3,  3,  3,  3,  3,  3,
```

```
53     4,  4,  4,  4,  4,  4,  4,  4,
54     5,  5,  5,  5,  5,  5,  5,  5,
55     6,  6,  6,  6,  6,  6,  6,  6,
56     7,  7,  7,  7,  7,  7,  7,  7,
57     8,  8,  8,  8,  8,  8,  8,  8,
58     9,  9,  9,  9,  9,  9,  9,  9,
59    10, 10, 10, 10, 10, 10, 10, 10,
60    11, 11, 11, 11, 11, 11, 11, 11,
61    12, 12, 12, 12, 12, 12, 12, 12,
62    13, 13, 13, 13, 13, 13, 13, 13,
63    14, 14, 14, 14, 14, 14, 14, 14,
64    15, 15, 15, 15, 15, 15, 15, 15,
65    16, 16, 16, 16, 16, 16, 16, 16};
```

```
67  __declspec(align(16)) short mmx_div[]={ // 放大 65536 倍, 为满足 MMX 指令, 重复成 4 倍大小
68      FIX(1),  FIX(1),  FIX(1),  FIX(1),
69      FIX(2),  FIX(2),  FIX(2),  FIX(2),
70      FIX(3),  FIX(3),  FIX(3),  FIX(3),
71      FIX(4),  FIX(4),  FIX(4),  FIX(4),
72      FIX(5),  FIX(5),  FIX(5),  FIX(5),
73      FIX(6),  FIX(6),  FIX(6),  FIX(6),
74      FIX(7),  FIX(7),  FIX(7),  FIX(7),
75      FIX(8),  FIX(8),  FIX(8),  FIX(8),
76      FIX(9),  FIX(9),  FIX(9),  FIX(9),
77      FIX(10), FIX(10), FIX(10), FIX(10),
78      FIX(11), FIX(11), FIX(11), FIX(11),
79      FIX(12), FIX(12), FIX(12), FIX(12),
80      FIX(13), FIX(13), FIX(13), FIX(13),
81      FIX(14), FIX(14), FIX(14), FIX(14),
82      FIX(15), FIX(15), FIX(15), FIX(15),
83      FIX(16), FIX(16), FIX(16), FIX(16),
84      FIX(17), FIX(17), FIX(17), FIX(17),
85      FIX(18), FIX(18), FIX(18), FIX(18),
86      FIX(19), FIX(19), FIX(19), FIX(19),
87      FIX(20), FIX(20), FIX(20), FIX(20),
88      FIX(21), FIX(21), FIX(21), FIX(21),
89      FIX(22), FIX(22), FIX(22), FIX(22),
90      FIX(23), FIX(23), FIX(23), FIX(23),
91      FIX(24), FIX(24), FIX(24), FIX(24),
92      FIX(25), FIX(25), FIX(25), FIX(25),
93      FIX(26), FIX(26), FIX(26), FIX(26),
94      FIX(27), FIX(27), FIX(27), FIX(27),
95      FIX(28), FIX(28), FIX(28), FIX(28),
96      FIX(29), FIX(29), FIX(29), FIX(29),
97      FIX(30), FIX(30), FIX(30), FIX(30),
98      FIX(31), FIX(31), FIX(31), FIX(31),
```

```
99     FIX(32), FIX(32), FIX(32), FIX(32));

101 uint32_t quant_h263_intra(int16_t* coeff, const int16_t* data, const uint32_t quant,
102     const uint32_t dcscalar)
103 {
104     __asm
105     {
106         mov esi, data      ; data

108         movsx eax, word ptr [esi]  ; data[0]

110         mov ecx, dcscalar    ; dcscalar
111         mov edx, eax
112         sar ecx, 1
113         add eax, ecx
114         sub edx, ecx
115         cmovl eax, edx      ; +/- dcscalar/2
116         mov ecx, quant     ; quant
117         cdq
118         idiv dword ptr [dcscalar]  ; dcscalar
119         mov edx, coeff     ; coeff
120         movdqu xmm7, [mmx_div+ecx * 8 - 8]

122         mov ecx, 2
123         movlhps xmm7, xmm7

125 loop1:

127         movdqa xmm0, [esi]
128         pxor xmm4, xmm4
129         movdqa xmm1, [esi + 16]
130         pcmpgtw xmm4, xmm0
131         pxor xmm5, xmm5
132         pmulhw xmm0, xmm7
133         pcmpgtw xmm5, xmm1
134         movdqa xmm2, [esi+32]
135         psubw xmm0, xmm4
136         pmulhw xmm1, xmm7
137         pxor xmm4, xmm4
138         movdqa xmm3, [esi+48]
139         pcmpgtw xmm4, xmm2
140         psubw xmm1, xmm5
141         pmulhw xmm2, xmm7
142         pxor xmm5, xmm5
143         pcmpgtw xmm5, xmm3
144         pmulhw xmm3, xmm7
```

```
145     psubw xmm2, xmm4
146     psubw xmm3, xmm5
147     movdqa [edx], xmm0
148     lea esi, [esi+64]
149     movdqa [edx + 16], xmm1
150     movdqa [edx + 32], xmm2
151     movdqa [edx + 48], xmm3

153     dec ecx
154     lea edx, [edx+64]
155     jne loop1

157     mov edx, coeff      ; coeff
158     mov [edx], ax
159     xor eax, eax        ; return 0
160 }
161 }
/* 第 115 行很巧妙的利用 cmovl 指令避开+/- dcscalar/2 判断运算。
   intra 量化正数量化只需要 pmulhw 一条核心指令。
   intra 量化负数量化多需要 pcmpgtw 和 psubw 两条核心指令。
   程序那么大是因为每次处理好几个，为减小指令数据流的依赖性，还打乱了逻辑。
// */

163 uint32_t quant_h263_inter(int16_t* coeff, const int16_t* data, const uint32_t quant)
164 {
165     __asm
166     {
167         mov edi, coeff      ; coeff
168         mov esi, data       ; data
169         mov eax, quant      ; quant

171         xor ecx, ecx

173         pxor xmm5, xmm5          ; sum

175         movq mm0, [mmx_sub + eax*8 - 8] ; sub
176         movq2dq xmm6, mm0         ; load into low 8 bytes
177         movlhps xmm6, xmm6        ; duplicate into high 8 bytes

179         movq mm0, [mmx_div + eax*8 - 8] ; divider
180         movq2dq xmm7, mm0
181         movlhps xmm7, xmm7

183 ALIGN 16
184 qes2_loop:
```

```
185     movdqa xmm0, [esi + ecx*8]           ; xmm0 = [1st]
186     movdqa xmm3, [esi + ecx*8 + 16]      ; xmm3 = [2nd]
187     pxor xmm1, xmm1
188     pxor xmm4, xmm4
189     pcmpgtw xmm1, xmm0
190     pcmpgtw xmm4, xmm3
191     pxor xmm0, xmm1
192     pxor xmm3, xmm4
193     psubw xmm0, xmm1
194     psubw xmm3, xmm4
195     psubusw xmm0, xmm6
196     psubusw xmm3, xmm6
197     pmulhw xmm0, xmm7
198     pmulhw xmm3, xmm7
199     paddw xmm5, xmm0
200     pxor xmm0, xmm1
201     paddw xmm5, xmm3
202     pxor xmm3, xmm4
203     psubw xmm0, xmm1
204     psubw xmm3, xmm4
205     movdqa [edi + ecx*8], xmm0
206     movdqa [edi + ecx*8 + 16], xmm3

208     add ecx, 4
209     cmp ecx, 16
210     jnz qes2_loop

212     movdqu xmm6, [plus_one]
213     pmaddwd xmm5, xmm6
214     movhlps xmm6, xmm5
215     paddd xmm5, xmm6
216     movdq2q mm0, xmm5

218     movq mm5, mm0
219     psrlq mm5, 32
220     paddd mm0, mm5

222     movd eax, mm0           ; return sum
223 }
224 }
/* inter 量化因为要处理修正值和计算量化系数和, 和 intra 比较要复杂一点点了。
// */

226 uint32_t dequant_h263_intra(int16_t* data, const int16_t* coeff,
227     const uint32_t quant, const uint32_t dscalar)
228 {
```



```
229     __asm
230     {
231         mov ecx, quant                ; quant
232         mov eax, coeff                ; coeff
233
234         movd xmm6, ecx                ; quant
235
236         shl ecx, 31
237         pshufbw xmm6, xmm6, 0
238         pcmpeqw xmm0, xmm0
239         movlhps xmm6, xmm6            ; all quant
240         movd xmm1, ecx
241         movdqa xmm5, xmm0
242         movdqa xmm7, xmm6
243         mov edx, data                 ; data
244         paddw xmm7, xmm7              ; quant *2
245         psllw xmm0, xmm1              ; quant & 1 ? 0 : - 1
246         movdqa xmm4, xmm5
247         paddw xmm6, xmm0              ; quant-1
248         psrlw xmm4, 5                 ; 2047
249         mov ecx, 4
250         pxor xmm5, xmm4               ; mm5=-2048
251
252 loop1:
253     movdqa xmm0, [eax]
254     pxor xmm2, xmm2
255     pxor xmm3, xmm3
256
257     cmpgtw xmm2, xmm0
258     pcmpeqw xmm3, xmm0
259     pmullw xmm0, xmm7                ; * 2 * quant
260     movdqa xmm1, [eax+16]
261
262     psubw xmm0, xmm2
263     pxor xmm2, xmm6
264     pandn xmm3, xmm2
265     pxor xmm2, xmm2
266     paddw xmm0, xmm3
267     pxor xmm3, xmm3
268     pcmpeqw xmm2, xmm1
269     cmpgtw xmm3, xmm1
270     pmullw xmm1, xmm7
271
272     pminsw xmm0, xmm4
273     psubw xmm1, xmm3
274     pxor xmm3, xmm6
```

```
275     pandn xmm2, xmm3
276     paddw xmm1, xmm2

278     pmaxsw xmm0, xmm5
279     pminsw xmm1, xmm4
280     movdqa [edx], xmm0
281     pmaxsw xmm1, xmm5
282     lea eax, [eax+32]
283     movdqa [edx+16], xmm1

285     dec ecx
286     lea edx, [edx+32]
287     jne loop1

289     ; deal with DC

291     mov eax, coeff           ; coeff
292     movsx eax, word ptr [eax]
293     imul dword ptr [dcscale] ; dcscale
294     mov edx, [data]          ; data
295     movd xmm0, eax
296     pminsw xmm0, xmm4
297     pmaxsw xmm0, xmm5
298     movd eax, xmm0

300     mov [edx], ax

302     xor eax, eax             ; return 0
303 }
304 }

306 uint32_t dequant_h263_inter(int16_t* data, const int16_t* coeff, const uint32_t quant)
307 {
308     __asm
309     {
310         mov ecx, quant           ; quant
311         mov eax, coeff           ; coeff

313         movdqu xmm6, [mmx_quant + ecx*8] ; quant
314         inc ecx
315         pcmpeqw xmm5, xmm5
316         and ecx, 1
317         movlhps xmm6, xmm6
318         movd xmm0, ecx
319         movdqa xmm7, xmm6
320         pshufhw xmm0, xmm0, 0
```

```
321      movdqa xmm4, xmm5
322      mov edx, data                ; data
323      movlhps xmm0, xmm0
324      paddw xmm7, xmm7
325      psubw xmm6, xmm0
326      psrlw xmm4, 5      ; 2047
327      mov ecx, 4
328      pxor xmm5, xmm4 ; mm5=-2048

330 loop1:
331      movdqa xmm0, [eax]
332      pxor xmm3, xmm3
333      pxor xmm2, xmm2
334      pcmpeqw xmm3, xmm0
335      pcmpgtw xmm2, xmm0
336      pmullw xmm0, xmm7      ; * 2 * quant
337      pandn xmm3, xmm6
338      movdqa xmm1, [eax+16]
339      psubw xmm0, xmm2
340      pxor xmm2, xmm3
341      pxor xmm3, xmm3
342      paddw xmm0, xmm2
343      pxor xmm2, xmm2
344      pcmpgtw xmm3, xmm1
345      pcmpeqw xmm2, xmm1
346      pmullw xmm1, xmm7
347      pandn xmm2, xmm6
348      psubw xmm1, xmm3
349      pxor xmm3, xmm2
350      paddw xmm1, xmm3

352      pminsw xmm0, xmm4
353      pminsw xmm1, xmm4
354      pmaxsw xmm0, xmm5
355      pmaxsw xmm1, xmm5

357      movdqa [edx], xmm0
358      lea eax, [eax+32]
359      movdqa [edx+16], xmm1

361      dec ecx
362      lea edx, [edx+32]
363      jne loop1

365      xor eax, eax                ; return 0
366      }
```

```
367 }
```

```
370 #else
```

```
373 static const uint32_t multipliers[32] =
374 {
375     0,      FIX(1),  FIX(2),  FIX(3),
376     FIX(4),  FIX(5),  FIX(6),  FIX(7),
377     FIX(8),  FIX(9),  FIX(10), FIX(11),
378     FIX(12), FIX(13), FIX(14), FIX(15),
379     FIX(16), FIX(17), FIX(18), FIX(19),
380     FIX(20), FIX(21), FIX(22), FIX(23),
381     FIX(24), FIX(25), FIX(26), FIX(27),
382     FIX(28), FIX(29), FIX(30), FIX(31)
383 };
```

/* 第 373 行到第 383 行是放大 65536 倍的量化表，改变量化时除法运算为乘法运算，也便于应用 PC 机汇编加速指令加速计算

```
// */
```

```
385 uint32_t quant_h263_intra(int16_t* coeff, const int16_t* data,
386     const uint32_t quant, const uint32_t dcscalar)
387 {
388     const uint32_t mult = multipliers[quant];
389     const uint16_t quant_m_2 = quant << 1;
390     int i;

392     coeff[0] = DIV_DIV(data[0], (int32_t) dcscalar);

394     for (i = 1; i < 64; i++)
395     {
396         int16_t acLevel = data[i];

398         if (acLevel < 0)
399         {
400             acLevel = -acLevel;
401             if (acLevel < quant_m_2)
402             {
403                 coeff[i] = 0;
404                 continue;
405             }
406             acLevel = (acLevel * mult) >> SCALEBITS;
407             coeff[i] = -acLevel;
408         }
    }
```

```
409         else
410         {
411             if (acLevel < quant_m_2)
412             {
413                 coeff[i] = 0;
414                 continue;
415             }
416             acLevel = (acLevel * mult) >> SCALEBITS;
417             coeff[i] = acLevel;
418         }
419     }

421     return(0);
422 }

/* 第 392 行量化 DC 系数。
   第 400 行 intra 块没有修正，直接量化。
   第 401 行和第 411 行判断是否大于 2 倍的量化系数，小则直接置 0
   第 406 行和第 416 行中的 >>SCALEBITS 是缩小 65536 倍，抵消量化表放大的 65536 倍
   // */

424 uint32_t quant_h263_inter(int16_t* coeff, const int16_t* data, const uint32_t quant)
425 {
426     const uint32_t mult = multipliers[quant];
427     const uint16_t quant_m_2 = quant << 1;
428     const uint16_t quant_d_2 = quant >> 1;
429     uint32_t sum = 0;
430     uint32_t i;

432     for (i = 0; i < 64; i++)
433     {
434         int16_t acLevel = data[i];

436         if (acLevel < 0)
437         {
438             acLevel = (-acLevel) - quant_d_2;
439             if (acLevel < quant_m_2)
440             {
441                 coeff[i] = 0;
442                 continue;
443             }

445             acLevel = (acLevel * mult) >> SCALEBITS;
446             sum += acLevel;
447             coeff[i] = -acLevel;
448         }
449         else
```

```
450     {
451         acLevel -= quant_d_2;
452         if (acLevel < quant_m_2)
453         {
454             coeff[i] = 0;
455             continue;
456         }
457         acLevel = (acLevel * mult) >> SCALEBITS;
458         sum += acLevel;
459         coeff[i] = acLevel;
460     }
461 }
```

```
463     return(sum);
```

```
464 }
```

/* 第 438 行是修正量化前的系数，而 intra 块不用修正。

第 446 行和第 458 行是量化的时候计算量化系数和，用于生成 cbp。因为 xvid 原始代码支持 AC_DC 预测，量化的时候无法判断 intra 是否使用 AC_DC 预测，因此无法在量化的时候计算 cbp。

```
// */
```

```
466 uint32_t dequant_h263_intra(int16_t* data, const int16_t* coeff,
467                             const uint32_t quant, const uint32_t dcscalar)
468 {
469     const int32_t quant_m_2 = quant << 1;
470     const int32_t quant_add = (quant & 1 ? quant : quant - 1);
471     int i;

473     data[0] = coeff[0] * dcscalar;
474     if (data[0] < -2048)
475         data[0] = -2048;
476     else if (data[0] > 2047)
477         data[0] = 2047;

479     for (i = 1; i < 64; i++)
480     {
481         int32_t acLevel = coeff[i];

483         if (acLevel == 0)
484         {
485             data[i] = 0;
486         }
487         else if (acLevel < 0)
488         {
489             acLevel = quant_m_2 * -acLevel + quant_add;
490             data[i] = (acLevel <= 2048 ? -acLevel : -2048);
491         }
```

```
492         else
493         {
494             acLevel = quant_m_2 * acLevel + quant_add;
495             data[i] = (acLevel <= 2047 ? acLevel : 2047);
496         }
497     }

499     return(0);
500 }

/* 第 470 行是奇偶控制。
   第 473 行到第 477 行, DC 系数反量化 和 饱和运算。
   第 479 行到第 497 行, AC 系数反量化 和 饱和运算。
// */

502 uint32_t dequant_h263_inter(int16_t* data, const int16_t* coeff, const uint32_t quant)
503 {
504     const uint16_t quant_m_2 = quant << 1;
505     const uint16_t quant_add = (quant & 1 ? quant : quant - 1);
506     int i;

508     for (i = 0; i < 64; i++)
509     {
510         int16_t acLevel = coeff[i];








512         if (acLevel == 0)
513         {
514             data[i] = 0;
515         }
516         else if (acLevel < 0)
517         {
518             acLevel = acLevel * quant_m_2 - quant_add;
519             data[i] = (acLevel >= -2048 ? acLevel : -2048);
520         }
521         else
522         {
523             acLevel = acLevel * quant_m_2 + quant_add;
524             data[i] = (acLevel <= 2047 ? acLevel : 2047);
525         }
526     }

528     return(0);
529 }

/* inter 块把 DC 系数混合进 AC 系数一致编码。程序流程更简单
// */
```

第八章 码流级程序

8.1 文件列表

类型	名称	大小
 h	cbp.h	106 bytes
 c	cbp.c	4447 bytes
 h	vlc_codes.h	23259 bytes
 h	mbcoding.h	427 bytes
 c	mbcoding.c	8844 bytes
 h	bitstream.h	2517 bytes
 c	bitstream.c	2774 bytes

8.2 cbp.h 文件

8.2.1 功能描述

计算 Intra 宏块 CBP 值的函数原型的声明，inter 宏块 CBP 值计算合并到量化函数中。因为 XVid 支持 Intra 宏块 AC 系数预测编码，要用量化系数进行比较来决定是水平 AC 预测还是垂直 AC 预测，所以在量化后计算方便。

主作者简化的 XVid 不支持 AC 系数预测，此时可以合并到量化中，各位读者可以自行优化修改。

8.2.2 文件注释

```
1  #ifndef _ENCODER_CBP_H_
2  #define _ENCODER_CBP_H_

4  uint32_t calc_cbp (const int16_t * codes);

6  #endif
```

8.3 cbp.c 文件

8.3.1 功能描述

计算 Intra 宏块 CBP 值的函数的实现，包括 C 语言和 PC 汇编语言版本。

8.3.2 文件注释

```
1  #include "../portab.h"
2  #include "cbp.h"

4  #ifdef CBP_ACC

6  __declspec(align(16)) short ignore_dc[8]={0, -1, -1, -1, -1, -1, -1, -1};
```



```
8  uint32_t calc_cbp(const int16_t codes[6 * 64])
9  {
10     __asm
11     {
12         mov edx, codes          ; coeff[]
13         xor eax, eax           ; cbp = 0

15         movdqa xmm7, [ignore_dc] ; mask to ignore dc value
16         pxor xmm6, xmm6       ; zero

18         movdqa xmm0, [edx]
19         pand xmm0, xmm7

21         movdqa xmm1, [edx+16]

23         por xmm0, [edx+32]     ; 异或运算来判断是否非零, 关键运算
24         por xmm1, [edx+48]
25         por xmm0, [edx+64]
26         por xmm1, [edx+80]
27         por xmm0, [edx+96]
28         por xmm1, [edx+112]

30         por xmm0, xmm1        ; xmm0 = xmm1 = 128 bits worth of info
31         psadbw xmm0, xmm6     ; contains 2 dwords with sums
32         movhlps xmm1, xmm0    ; move high dword from xmm0 to low xmm1
33         por xmm0, xmm1        ; combine
34         movd ecx, xmm0        ; if ecx set, values were found
/* 第 30 行计算非零信息, 第 31 行以字节为单位求和, 注意 psadbw 结果保存在 bit[15...0]和
   bit[80...64]中, 所以要用第 32 行从高位移到低位, 经过第 33 行再次异或后得到所有非
   零信息, 转送到 ecx 后测试是否为零。

// */
36         test ecx, ecx
37         jz blk2
38         or eax, (1<<5)

40 blk2:
41         movdqa xmm0, [edx+128]
42         pand xmm0, xmm7
43         movdqa xmm1, [edx+128+16]
```

```
45     por xmm0, [edx+128+32]
46     por xmm1, [edx+128+48]
47     por xmm0, [edx+128+64]
48     por xmm1, [edx+128+80]
49     por xmm0, [edx+128+96]
50     por xmm1, [edx+128+112]

52     por xmm0, xmm1      ; xmm0 = xmm1 = 128 bits worth of info
53     psadbw xmm0, xmm6   ; contains 2 dwords with sums
54     movhlps xmm1, xmm0  ; move high dword from xmm0 to low xmm1
55     por xmm0, xmm1      ; combine
56     movd ecx, xmm0      ; if ecx set, values were found

58     test ecx, ecx
59     jz blk3
60     or eax, (1<<4)

62 blk3:
63     movdqa xmm0, [edx+256]
64     pand xmm0, xmm7
65     movdqa xmm1, [edx+256+16]

67     por xmm0, [edx+256+32]
68     por xmm1, [edx+256+48]
69     por xmm0, [edx+256+64]
70     por xmm1, [edx+256+80]
71     por xmm0, [edx+256+96]
72     por xmm1, [edx+256+112]

74     por xmm0, xmm1      ; xmm0 = xmm1 = 128 bits worth of info
75     psadbw xmm0, xmm6   ; contains 2 dwords with sums
76     movhlps xmm1, xmm0  ; move high dword from xmm0 to low xmm1
77     por xmm0, xmm1      ; combine
78     movd ecx, xmm0      ; if ecx set, values were found

80     test ecx, ecx
81     jz blk4
82     or eax, (1<<3)
```

```
84 blk4:
85     movdqa xmm0, [edx+384]
86     pand xmm0, xmm7
87     movdqa xmm1, [edx+384+16]

89     por xmm0, [edx+384+32]
90     por xmm1, [edx+384+48]
91     por xmm0, [edx+384+64]
92     por xmm1, [edx+384+80]
93     por xmm0, [edx+384+96]
94     por xmm1, [edx+384+112]

96     por xmm0, xmm1      ; xmm0 = xmm1 = 128 bits worth of info
97     psadbw xmm0, xmm6   ; contains 2 dwords with sums
98     movhlps xmm1, xmm0  ; move high dword from xmm0 to low xmm1
99     por xmm0, xmm1      ; combine
100    movd ecx, xmm0       ; if ecx set, values were found

102    test ecx, ecx
103    jz blk5
104    or eax, (1<<2)

106 blk5:
107    movdqa xmm0, [edx+512]
108    pand xmm0, xmm7
109    movdqa xmm1, [edx+512+16]

111    por xmm0, [edx+512+32]
112    por xmm1, [edx+512+48]
113    por xmm0, [edx+512+64]
114    por xmm1, [edx+512+80]
115    por xmm0, [edx+512+96]
116    por xmm1, [edx+512+112]

118    por xmm0, xmm1      ; xmm0 = xmm1 = 128 bits worth of info
119    psadbw xmm0, xmm6   ; contains 2 dwords with sums
120    movhlps xmm1, xmm0  ; move high dword from xmm0 to low xmm1
121    por xmm0, xmm1      ; combine
122    movd ecx, xmm0       ; if ecx set, values were found
```

```
124         test ecx, ecx
125         jz blk6
126         or eax, (1<<1)

128 blk6:
129         movdqa xmm0, [edx+640]
130         pand xmm0, xmm7
131         movdqa xmm1, [edx+640+16]

133         por xmm0, [edx+640+32]
134         por xmm1, [edx+640+48]
135         por xmm0, [edx+640+64]
136         por xmm1, [edx+640+80]
137         por xmm0, [edx+640+96]
138         por xmm1, [edx+640+112]

140         por xmm0, xmm1           ; xmm0 = xmm1 = 128 bits worth of info
141         psadbw xmm0, xmm6        ; contains 2 dwords with sums
142         movhlps xmm1, xmm0       ; move high dword from xmm0 to low xmm1
143         por xmm0, xmm1           ; combine
144         movd ecx, xmm0           ; if ecx set, values were found

146         test ecx, ecx
147         jz blk7
148         or eax, (1<<0)
149 blk7:
150     }
151 }

153 #else

155 uint32_t calc_cbp(const int16_t codes[6 * 64])
156 {
157     unsigned int i=6;
158     uint32_t cbp = 0;

160     do {
161         uint64_t *codes64 = (uint64_t*) codes;
162         uint32_t *codes32 = (uint32_t*) codes;
```

```
164         cbp += cbp;
165         if (codes[1] || codes32[1])
166         {
167             cbp++;
168         }
169         else if (codes64[1] | codes64[2] | codes64[3])
170         {
171             cbp++;
172         }
173         else if (codes64[4] | codes64[5] | codes64[6] | codes64[7])
174         {
175             cbp++;
176         }
177         else if (codes64[8] | codes64[9] | codes64[10] | codes64[11])
178         {
179             cbp++;
180         }
181         else if (codes64[12] | codes64[13] | codes64[14] | codes64[15])
182         {
183             cbp++;
184         }
185         codes += 64;
186         i--;
187     } while (i != 0);

189     return cbp;
190 }
```

/* 计算宏块 cbp 值，每一次 do{}while() 循环计算一个 8x8 小块

如果按照从上到下，从左到右的顺序从 0 开始排 8x8 小块的像素点，第 165 行到第 168 行比较计算 1 到 3 号像素点。第 169 行到第 172 行比较计算 4 到 15 号像素点。第 173 行到第 176 行比较计算 16 到 31 号像素点。第 177 行到第 180 行比较计算 32 到 47 号像素点。第 181 行到第 184 行比较计算 48 到 63 号像素点。

因为计算 cbp 只需要判断是否有非零系数，不判断大小和位置，所以可以采用这种 if()else() 结构提前跳出循环。

此函数的精妙之处在于第 164 行，用加法实现移位。

```
// */
192 #endif
```

8.4 vlc_codes.h 文件

8.4.1 功能描述

各种 VLC 表的数组描述，注意为了做程序方便，很多 VLC 表做了特别处理，因此和标准中的表模样有些不一样，具体解释请仔细阅读理解此文件中的注释，理解的 VLC 表数组就理解了 huffman 编码函数的算法。

8.4.2 文件注释

```
1  #ifndef _VLC_CODES_H_
2  #define _VLC_CODES_H_

4  #define ESCAPE  3
5  #define ESCAPE1 6
6  #define ESCAPE2 14
7  #define ESCAPE3 15

9  typedef struct
10 {
11     uint32_t code;
12     uint8_t len;
13 }VLC;

15 typedef struct
16 {
17     uint8_t last;
18     uint8_t run;
19     int8_t level;
20 }EVENT;

22 typedef struct
23 {
24     VLC vlc;
25     EVENT event;
26 }VLC_TABLE;

28 static const uint16_t zigzag[64] = {
29     0,  1,  8, 16,  9,  2,  3, 10,
30     17, 24, 32, 25, 18, 11,  4,  5,
31     12, 19, 26, 33, 40, 48, 41, 34,
32     27, 20, 13,  6,  7, 14, 21, 28,
33     35, 42, 49, 56, 57, 50, 43, 36,
34     29, 22, 15, 23, 30, 37, 44, 51,
35     58, 59, 52, 45, 38, 31, 39, 46,
36     53, 60, 61, 54, 47, 55, 62, 63
37 };
```

/* 第 28 行到第 37 行是 zigzag 扫描表，不支持 AC 预测，仅 DC 预测，因此不需要水平扫描表和垂直扫


```

49      {2, 4}, {6, 7}, {4, 7}, {3, 8}, {3, 9}, {0, 0}, {0, 0}, {0, 0},
49      {5, 6}, {5, 9}, {5, 8}, {3, 7}, {2, 9}
50      };

```

/* 第46行到第50行是 inter 块使用的 cbpc, 计算 $(mbtype \& 7) \mid ((cbpc \& 3) \ll 3)$ 值并以此值排序。

P-VOP 中 mcbpc 的变长码表			转换方法和转换值
Code	mb type	cbpc (56)	$(mbtype \& 7) \mid ((cbpc \& 3) \ll 3)$
1	0	00	00000 (0)
0011	0	01	01000 (8)
0010	0	10	10000 (16)
0001 01	0	11	11000 (24)
011	1	00	00001 (1)
0000 111	1	01	01001 (9)
0000 110	1	10	10001 (17)
0000 0010 1	1	11	11001 (25)
010	2	00	00010 (2)
0000 101	2	01	01010 (10)
0000 100	2	10	10010 (18)
0000 0101	2	11	11010 (26)
0001 1	3	00	00011 (3)
0000 0100	3	01	01011 (11)
0000 0011	3	10	10011 (19)
0000 011	3	11	11011 (27)
0001 00	4	00	00100 (4)
0000 0010 0	4	01	01100 (12)
0000 0001 1	4	10	10100 (20)
0000 0001 0	4	11	11100 (28)
0000 0000 1	Stuffing	--	

```
// */
```

```

53 const VLC xvid_cbpy_tab[16] = {
54     {3, 4}, {5, 5}, {4, 5}, {9, 4}, {3, 5}, {7, 4}, {2, 6}, {11, 4},
55     {2, 5}, {3, 6}, {5, 4}, {10, 4}, {4, 4}, {8, 4}, {6, 4}, {3, 2} };

```

/* 第53行到第55行 cbpy VLC 数组由 I-VOP cbpy 变长码表转换而来, intra 和 inter 是 16 补数关系。
简单优化方法, 可以使用 P-VOP cbpy VLC 表, I-VOP cbpy 用 16 补数算法计算出来。

I-VOP 和 P-VOP 中 cbpy 的变长码表		
Code	Intra (1234)	Inter (1234)
0011	0000	1111
0010 1	0001	1110
0010 0	0010	1101
1001	0011	1100
0001 1	0100	1011
0111	0101	1010
0000 10	0110	1001
1011	0111	1000
0001 0	1000	0111
0000 11	1001	0110
0101	1010	0101
1010	1011	0100
0100	1100	0011

1000	1101	0010
0110	1110	0001
11	1111	0000

// */

58 const VLC mb_motion_table[65] = {

59 {0x05, 13}, {0x07, 13}, //

60 {0x05, 12}, {0x07, 12}, //

61 {0x09, 12}, {0x0b, 12}, //

62 {0x0d, 12}, {0x0f, 12}, //

63 {0x09, 11}, {0x0b, 11}, //

64 {0x0d, 11}, {0x0f, 11}, //

65 {0x11, 11}, {0x13, 11}, //

66 {0x15, 11}, {0x17, 11}, //

67 {0x19, 11}, {0x1b, 11}, //

68 {0x1d, 11}, {0x1f, 11}, //

69 {0x21, 11}, {0x23, 11}, //

70 {0x13, 10}, {0x15, 10}, //

71 {0x17, 10}, {0x07, 8}, //

72 {0x09, 8}, {0x0b, 8}, //

73 {0x07, 7}, {0x03, 5}, //

74 {0x03, 4}, {0x03, 3}, //

75 {0x01, 1}, {0x02, 3}, //

76 {0x02, 4}, {0x02, 5}, //

77 {0x06, 7}, {0x0a, 8}, //

78 {0x08, 8}, {0x06, 8}, //

79 {0x16, 10}, {0x14, 10}, //

80 {0x12, 10}, {0x22, 11}, //

81 {0x20, 11}, {0x1e, 11}, //

82 {0x1c, 11}, {0x1a, 11}, //

83 {0x18, 11}, {0x16, 11}, //

84 {0x14, 11}, {0x12, 11}, //

85 {0x10, 11}, {0x0e, 11}, //

86 {0x0c, 11}, {0x0a, 11}, //

87 {0x08, 11}, {0x0e, 12}, //

88 {0x0c, 12}, {0x0a, 12}, //

89 {0x08, 12}, {0x06, 12}, //

90 {0x04, 12}, {0x06, 13}, //

91 {0x04, 13} //

92 };

/* 第 58 行到第 92 行 motion VLC 数组直接由 MP4 标准 mvd 变长码表而来, 没什么技巧。

// */

94 const VLC dcy_tab[511] = {

95 {0x100, 15}, {0x101, 15}, {0x102, 15}, {0x103, 15},

96 {0x104, 15}, {0x105, 15}, {0x106, 15}, {0x107, 15},

Vector Codes	diff
0000 0000 0010 1	-16
0000 0000 0011 1	-15.5
0000 0000 0101	-15
0000 0000 0111	-14.5
0000 0000 1001	-14
0000 0000 1011	-13.5
0000 0000 1101	-13
0000 0000 1111	-12.5
0000 0001 001	-12
0000 0001 011	-11.5
0000 0001 101	-11
0000 0001 111	-10.5
0000 0010 001	-10
0000 0010 011	-9.5
0000 0010 101	-9
0000 0010 111	-8.5
0000 0011 001	-8
0000 0011 011	-7.5
0000 0011 101	-7
0000 0011 111	-6.5
0000 0100 001	-6
0000 0100 011	-5.5
0000 0100 11	-5
0000 0101 01	-4.5
0000 0101 11	-4
0000 0111	-3.5
0000 1001	-3
0000 1011	-2.5
0000 111	-2
0001 1	-1.5
0011	-1
011	-0.5
1	0

Vector Codes	diff
0000 0000 0010 0	16
0000 0000 0011 0	15.5
0000 0000 0100	15
0000 0000 0110	14.5
0000 0000 1000	14
0000 0000 1010	13.5
0000 0000 1100	13
0000 0000 1110	12.5
0000 0001 000	12
0000 0001 010	11.5
0000 0001 100	11
0000 0001 110	10.5
0000 0010 000	10
0000 0010 010	9.5
0000 0010 100	9
0000 0010 110	8.5
0000 0011 000	8
0000 0011 010	7.5
0000 0011 100	7
0000 0011 110	6.5
0000 0100 000	6
0000 0100 010	5.5
0000 0100 10	5
0000 0101 00	4.5
0000 0101 10	4
0000 0110	3.5
0000 1000	3
0000 1010	2.5
0000 110	2
0001 0	1.5
0010	1
010	0.5
1	0

```
97      {0x108, 15}, {0x109, 15}, {0x10a, 15}, {0x10b, 15},
98      {0x10c, 15}, {0x10d, 15}, {0x10e, 15}, {0x10f, 15},
99      {0x110, 15}, {0x111, 15}, {0x112, 15}, {0x113, 15},
100     {0x114, 15}, {0x115, 15}, {0x116, 15}, {0x117, 15},
101     {0x118, 15}, {0x119, 15}, {0x11a, 15}, {0x11b, 15},
102     {0x11c, 15}, {0x11d, 15}, {0x11e, 15}, {0x11f, 15},
103     {0x120, 15}, {0x121, 15}, {0x122, 15}, {0x123, 15},
104     {0x124, 15}, {0x125, 15}, {0x126, 15}, {0x127, 15},
105     {0x128, 15}, {0x129, 15}, {0x12a, 15}, {0x12b, 15},
106     {0x12c, 15}, {0x12d, 15}, {0x12e, 15}, {0x12f, 15},
107     {0x130, 15}, {0x131, 15}, {0x132, 15}, {0x133, 15},
108     {0x134, 15}, {0x135, 15}, {0x136, 15}, {0x137, 15},
109     {0x138, 15}, {0x139, 15}, {0x13a, 15}, {0x13b, 15},
110     {0x13c, 15}, {0x13d, 15}, {0x13e, 15}, {0x13f, 15},
111     {0x140, 15}, {0x141, 15}, {0x142, 15}, {0x143, 15},
112     {0x144, 15}, {0x145, 15}, {0x146, 15}, {0x147, 15},
113     {0x148, 15}, {0x149, 15}, {0x14a, 15}, {0x14b, 15},
114     {0x14c, 15}, {0x14d, 15}, {0x14e, 15}, {0x14f, 15},
115     {0x150, 15}, {0x151, 15}, {0x152, 15}, {0x153, 15},
116     {0x154, 15}, {0x155, 15}, {0x156, 15}, {0x157, 15},
117     {0x158, 15}, {0x159, 15}, {0x15a, 15}, {0x15b, 15},
118     {0x15c, 15}, {0x15d, 15}, {0x15e, 15}, {0x15f, 15},
119     {0x160, 15}, {0x161, 15}, {0x162, 15}, {0x163, 15},
120     {0x164, 15}, {0x165, 15}, {0x166, 15}, {0x167, 15},
121     {0x168, 15}, {0x169, 15}, {0x16a, 15}, {0x16b, 15},
122     {0x16c, 15}, {0x16d, 15}, {0x16e, 15}, {0x16f, 15},
123     {0x170, 15}, {0x171, 15}, {0x172, 15}, {0x173, 15},
124     {0x174, 15}, {0x175, 15}, {0x176, 15}, {0x177, 15},
125     {0x178, 15}, {0x179, 15}, {0x17a, 15}, {0x17b, 15},
126     {0x17c, 15}, {0x17d, 15}, {0x17e, 15}, {0x17f, 15},
127     {0x80, 13}, {0x81, 13}, {0x82, 13}, {0x83, 13},
128     {0x84, 13}, {0x85, 13}, {0x86, 13}, {0x87, 13},
129     {0x88, 13}, {0x89, 13}, {0x8a, 13}, {0x8b, 13},
130     {0x8c, 13}, {0x8d, 13}, {0x8e, 13}, {0x8f, 13},
131     {0x90, 13}, {0x91, 13}, {0x92, 13}, {0x93, 13},
132     {0x94, 13}, {0x95, 13}, {0x96, 13}, {0x97, 13},
133     {0x98, 13}, {0x99, 13}, {0x9a, 13}, {0x9b, 13},
134     {0x9c, 13}, {0x9d, 13}, {0x9e, 13}, {0x9f, 13},
135     {0xa0, 13}, {0xa1, 13}, {0xa2, 13}, {0xa3, 13},
136     {0xa4, 13}, {0xa5, 13}, {0xa6, 13}, {0xa7, 13},
137     {0xa8, 13}, {0xa9, 13}, {0xaa, 13}, {0xab, 13},
138     {0xac, 13}, {0xad, 13}, {0xae, 13}, {0xaf, 13},
139     {0xb0, 13}, {0xb1, 13}, {0xb2, 13}, {0xb3, 13},
140     {0xb4, 13}, {0xb5, 13}, {0xb6, 13}, {0xb7, 13},
141     {0xb8, 13}, {0xb9, 13}, {0xba, 13}, {0xbb, 13},
142     {0xbc, 13}, {0xbd, 13}, {0xbe, 13}, {0xbf, 13},
```

```
143     {0x40, 11}, {0x41, 11}, {0x42, 11}, {0x43, 11},
144     {0x44, 11}, {0x45, 11}, {0x46, 11}, {0x47, 11},
145     {0x48, 11}, {0x49, 11}, {0x4a, 11}, {0x4b, 11},
146     {0x4c, 11}, {0x4d, 11}, {0x4e, 11}, {0x4f, 11},
147     {0x50, 11}, {0x51, 11}, {0x52, 11}, {0x53, 11},
148     {0x54, 11}, {0x55, 11}, {0x56, 11}, {0x57, 11},
149     {0x58, 11}, {0x59, 11}, {0x5a, 11}, {0x5b, 11},
150     {0x5c, 11}, {0x5d, 11}, {0x5e, 11}, {0x5f, 11},
151     {0x20, 9}, {0x21, 9}, {0x22, 9}, {0x23, 9},
152     {0x24, 9}, {0x25, 9}, {0x26, 9}, {0x27, 9},
153     {0x28, 9}, {0x29, 9}, {0x2a, 9}, {0x2b, 9},
154     {0x2c, 9}, {0x2d, 9}, {0x2e, 9}, {0x2f, 9},
155     {0x10, 7}, {0x11, 7}, {0x12, 7}, {0x13, 7},
156     {0x14, 7}, {0x15, 7}, {0x16, 7}, {0x17, 7},
157     {0x10, 6}, {0x11, 6}, {0x12, 6}, {0x13, 6},
158     {0x08, 4}, {0x09, 4}, {0x06, 3}, {0x03, 3},
159     {0x07, 3}, {0x0a, 4}, {0x0b, 4}, {0x14, 6},
160     {0x15, 6}, {0x16, 6}, {0x17, 6}, {0x18, 7},
161     {0x19, 7}, {0x1a, 7}, {0x1b, 7}, {0x1c, 7},
162     {0x1d, 7}, {0x1e, 7}, {0x1f, 7}, {0x30, 9},
163     {0x31, 9}, {0x32, 9}, {0x33, 9}, {0x34, 9},
164     {0x35, 9}, {0x36, 9}, {0x37, 9}, {0x38, 9},
165     {0x39, 9}, {0x3a, 9}, {0x3b, 9}, {0x3c, 9},
166     {0x3d, 9}, {0x3e, 9}, {0x3f, 9}, {0x60, 11},
167     {0x61, 11}, {0x62, 11}, {0x63, 11}, {0x64, 11},
168     {0x65, 11}, {0x66, 11}, {0x67, 11}, {0x68, 11},
169     {0x69, 11}, {0x6a, 11}, {0x6b, 11}, {0x6c, 11},
170     {0x6d, 11}, {0x6e, 11}, {0x6f, 11}, {0x70, 11},
171     {0x71, 11}, {0x72, 11}, {0x73, 11}, {0x74, 11},
172     {0x75, 11}, {0x76, 11}, {0x77, 11}, {0x78, 11},
173     {0x79, 11}, {0x7a, 11}, {0x7b, 11}, {0x7c, 11},
174     {0x7d, 11}, {0x7e, 11}, {0x7f, 11}, {0xc0, 13},
175     {0xc1, 13}, {0xc2, 13}, {0xc3, 13}, {0xc4, 13},
176     {0xc5, 13}, {0xc6, 13}, {0xc7, 13}, {0xc8, 13},
177     {0xc9, 13}, {0xca, 13}, {0xcb, 13}, {0xcc, 13},
178     {0xcd, 13}, {0xce, 13}, {0xcf, 13}, {0xd0, 13},
179     {0xd1, 13}, {0xd2, 13}, {0xd3, 13}, {0xd4, 13},
180     {0xd5, 13}, {0xd6, 13}, {0xd7, 13}, {0xd8, 13},
181     {0xd9, 13}, {0xda, 13}, {0xdb, 13}, {0xdc, 13},
182     {0xdd, 13}, {0xde, 13}, {0xdf, 13}, {0xe0, 13},
183     {0xe1, 13}, {0xe2, 13}, {0xe3, 13}, {0xe4, 13},
184     {0xe5, 13}, {0xe6, 13}, {0xe7, 13}, {0xe8, 13},
185     {0xe9, 13}, {0xea, 13}, {0xeb, 13}, {0xec, 13},
186     {0xed, 13}, {0xee, 13}, {0xef, 13}, {0xf0, 13},
187     {0xf1, 13}, {0xf2, 13}, {0xf3, 13}, {0xf4, 13},
188     {0xf5, 13}, {0xf6, 13}, {0xf7, 13}, {0xf8, 13},
```

```

189     {0xf9, 13}, {0xfa, 13}, {0xfb, 13}, {0xfc, 13},
190     {0xfd, 13}, {0xfe, 13}, {0xff, 13}, {0x180, 15},
191     {0x181, 15}, {0x182, 15}, {0x183, 15}, {0x184, 15},
192     {0x185, 15}, {0x186, 15}, {0x187, 15}, {0x188, 15},
193     {0x189, 15}, {0x18a, 15}, {0x18b, 15}, {0x18c, 15},
194     {0x18d, 15}, {0x18e, 15}, {0x18f, 15}, {0x190, 15},
195     {0x191, 15}, {0x192, 15}, {0x193, 15}, {0x194, 15},
196     {0x195, 15}, {0x196, 15}, {0x197, 15}, {0x198, 15},
197     {0x199, 15}, {0x19a, 15}, {0x19b, 15}, {0x19c, 15},
198     {0x19d, 15}, {0x19e, 15}, {0x19f, 15}, {0x1a0, 15},
199     {0x1a1, 15}, {0x1a2, 15}, {0x1a3, 15}, {0x1a4, 15},
200     {0x1a5, 15}, {0x1a6, 15}, {0x1a7, 15}, {0x1a8, 15},
201     {0x1a9, 15}, {0x1aa, 15}, {0x1ab, 15}, {0x1ac, 15},
202     {0x1ad, 15}, {0x1ae, 15}, {0x1af, 15}, {0x1b0, 15},
203     {0x1b1, 15}, {0x1b2, 15}, {0x1b3, 15}, {0x1b4, 15},
204     {0x1b5, 15}, {0x1b6, 15}, {0x1b7, 15}, {0x1b8, 15},
205     {0x1b9, 15}, {0x1ba, 15}, {0x1bb, 15}, {0x1bc, 15},
206     {0x1bd, 15}, {0x1be, 15}, {0x1bf, 15}, {0x1c0, 15},
207     {0x1c1, 15}, {0x1c2, 15}, {0x1c3, 15}, {0x1c4, 15},
208     {0x1c5, 15}, {0x1c6, 15}, {0x1c7, 15}, {0x1c8, 15},
209     {0x1c9, 15}, {0x1ca, 15}, {0x1cb, 15}, {0x1cc, 15},
210     {0x1cd, 15}, {0x1ce, 15}, {0x1cf, 15}, {0x1d0, 15},
211     {0x1d1, 15}, {0x1d2, 15}, {0x1d3, 15}, {0x1d4, 15},
212     {0x1d5, 15}, {0x1d6, 15}, {0x1d7, 15}, {0x1d8, 15},
213     {0x1d9, 15}, {0x1da, 15}, {0x1db, 15}, {0x1dc, 15},
214     {0x1dd, 15}, {0x1de, 15}, {0x1df, 15}, {0x1e0, 15},
215     {0x1e1, 15}, {0x1e2, 15}, {0x1e3, 15}, {0x1e4, 15},
216     {0x1e5, 15}, {0x1e6, 15}, {0x1e7, 15}, {0x1e8, 15},
217     {0x1e9, 15}, {0x1ea, 15}, {0x1eb, 15}, {0x1ec, 15},
218     {0x1ed, 15}, {0x1ee, 15}, {0x1ef, 15}, {0x1f0, 15},
219     {0x1f1, 15}, {0x1f2, 15}, {0x1f3, 15}, {0x1f4, 15},
220     {0x1f5, 15}, {0x1f6, 15}, {0x1f7, 15}, {0x1f8, 15},
221     {0x1f9, 15}, {0x1fa, 15}, {0x1fb, 15}, {0x1fc, 15},
222     {0x1fd, 15}, {0x1fe, 15}, {0x1ff, 15},
223 };

```

/* 第 94 行到第 223 行是 DC 系数数组, 把 $[-512, 511]$ 之间有效的差分 DC 系数按照规定计算 dc_size VLC 码和 DC differences, 然后再合成一个大 VLC 表。

举例 1: 比如 Diff DC 为 0 时, 查差分 DC 码表得到 Size 大小为 0 对应到 dc_size VLC 表为 011, 查差分 DC 码表 Additional code 空白即不用拼接, 所以是 {0x3, 3};

举例 2: 比如 Diff DC 为 1 时, 查差分 DC 码表得到 Size 大小为 1 对应到 dc_size VLC 表为 11, 查差分 DC 码表 Additional code 为 1, 拼接起来是 111, 所以是 {0x7, 3};

举例 3: 比如 Diff DC 为 -2 时, 查差分 DC 码表得到 Size 大小为 2 对应到 dc_size VLC 表为 10, 查差分 DC 码表 Additional code 为 01, 拼接起来是 1001, 所以是 {0x09, 4};

dct_dc_size_luminance 的变长码表	
Variable length code	dct_dc_size
011	0
11	1
10	2
010	3
001	4
0001	5
0000 1	6
0000 01	7
0000 001	8
0000 0001	9
0000 0000 1	10
0000 0000 01	11
0000 0000 001	12

差分 DC 码表		
Additional code	Diff DC	Size
000000000 to 011111111 *	-511 to -256	9
000000000 to 01111111	-255 to -128	8
00000000 to 0111111	-127 to -64	7
0000000 to 011111	-63 to -32	6
000000 to 01111	-31 to -16	5
00000 to 0111	-15 to -8	4
000 to 011	-7 to -4	3
00 to 01	-3 to -2	2
0	-1	1
	0	0
1	1	1
10 to 11	2 to 3	2
100 to 111	4 to 7	3
1000 to 1111	8 to 15	4
10000 to 11111	16 to 31	5
100000 to 111111	32 to 63	6
1000000 to 1111111	64 to 127	7
10000000 to 11111111	128 to 255	8
100000000 to 111111111 *	256 to 511	9

```
// */
```

```

225 const VLC dcc_tab[511] = { // 和 dcy_tab[]原理方法一样, 参见 dcy_tab[]的解释
226     {0x100, 16}, {0x101, 16}, {0x102, 16}, {0x103, 16},
227     {0x104, 16}, {0x105, 16}, {0x106, 16}, {0x107, 16},
228     {0x108, 16}, {0x109, 16}, {0x10a, 16}, {0x10b, 16},
229     {0x10c, 16}, {0x10d, 16}, {0x10e, 16}, {0x10f, 16},
230     {0x110, 16}, {0x111, 16}, {0x112, 16}, {0x113, 16},
231     {0x114, 16}, {0x115, 16}, {0x116, 16}, {0x117, 16},
232     {0x118, 16}, {0x119, 16}, {0x11a, 16}, {0x11b, 16},
233     {0x11c, 16}, {0x11d, 16}, {0x11e, 16}, {0x11f, 16},
234     {0x120, 16}, {0x121, 16}, {0x122, 16}, {0x123, 16},
235     {0x124, 16}, {0x125, 16}, {0x126, 16}, {0x127, 16},
236     {0x128, 16}, {0x129, 16}, {0x12a, 16}, {0x12b, 16},
237     {0x12c, 16}, {0x12d, 16}, {0x12e, 16}, {0x12f, 16},
238     {0x130, 16}, {0x131, 16}, {0x132, 16}, {0x133, 16},
239     {0x134, 16}, {0x135, 16}, {0x136, 16}, {0x137, 16},
240     {0x138, 16}, {0x139, 16}, {0x13a, 16}, {0x13b, 16},
241     {0x13c, 16}, {0x13d, 16}, {0x13e, 16}, {0x13f, 16},
242     {0x140, 16}, {0x141, 16}, {0x142, 16}, {0x143, 16},
243     {0x144, 16}, {0x145, 16}, {0x146, 16}, {0x147, 16},
244     {0x148, 16}, {0x149, 16}, {0x14a, 16}, {0x14b, 16},

```

```
245     {0x14c, 16}, {0x14d, 16}, {0x14e, 16}, {0x14f, 16},
246     {0x150, 16}, {0x151, 16}, {0x152, 16}, {0x153, 16},
247     {0x154, 16}, {0x155, 16}, {0x156, 16}, {0x157, 16},
248     {0x158, 16}, {0x159, 16}, {0x15a, 16}, {0x15b, 16},
249     {0x15c, 16}, {0x15d, 16}, {0x15e, 16}, {0x15f, 16},
250     {0x160, 16}, {0x161, 16}, {0x162, 16}, {0x163, 16},
251     {0x164, 16}, {0x165, 16}, {0x166, 16}, {0x167, 16},
252     {0x168, 16}, {0x169, 16}, {0x16a, 16}, {0x16b, 16},
253     {0x16c, 16}, {0x16d, 16}, {0x16e, 16}, {0x16f, 16},
254     {0x170, 16}, {0x171, 16}, {0x172, 16}, {0x173, 16},
255     {0x174, 16}, {0x175, 16}, {0x176, 16}, {0x177, 16},
256     {0x178, 16}, {0x179, 16}, {0x17a, 16}, {0x17b, 16},
257     {0x17c, 16}, {0x17d, 16}, {0x17e, 16}, {0x17f, 16},
258     {0x80, 14}, {0x81, 14}, {0x82, 14}, {0x83, 14},
259     {0x84, 14}, {0x85, 14}, {0x86, 14}, {0x87, 14},
260     {0x88, 14}, {0x89, 14}, {0x8a, 14}, {0x8b, 14},
261     {0x8c, 14}, {0x8d, 14}, {0x8e, 14}, {0x8f, 14},
262     {0x90, 14}, {0x91, 14}, {0x92, 14}, {0x93, 14},
263     {0x94, 14}, {0x95, 14}, {0x96, 14}, {0x97, 14},
264     {0x98, 14}, {0x99, 14}, {0x9a, 14}, {0x9b, 14},
265     {0x9c, 14}, {0x9d, 14}, {0x9e, 14}, {0x9f, 14},
266     {0xa0, 14}, {0xa1, 14}, {0xa2, 14}, {0xa3, 14},
267     {0xa4, 14}, {0xa5, 14}, {0xa6, 14}, {0xa7, 14},
268     {0xa8, 14}, {0xa9, 14}, {0xaa, 14}, {0xab, 14},
269     {0xac, 14}, {0xad, 14}, {0xae, 14}, {0xaf, 14},
270     {0xb0, 14}, {0xb1, 14}, {0xb2, 14}, {0xb3, 14},
271     {0xb4, 14}, {0xb5, 14}, {0xb6, 14}, {0xb7, 14},
272     {0xb8, 14}, {0xb9, 14}, {0xba, 14}, {0xbb, 14},
273     {0xbc, 14}, {0xbd, 14}, {0xbe, 14}, {0xbf, 14},
274     {0x40, 12}, {0x41, 12}, {0x42, 12}, {0x43, 12},
275     {0x44, 12}, {0x45, 12}, {0x46, 12}, {0x47, 12},
276     {0x48, 12}, {0x49, 12}, {0x4a, 12}, {0x4b, 12},
277     {0x4c, 12}, {0x4d, 12}, {0x4e, 12}, {0x4f, 12},
278     {0x50, 12}, {0x51, 12}, {0x52, 12}, {0x53, 12},
279     {0x54, 12}, {0x55, 12}, {0x56, 12}, {0x57, 12},
280     {0x58, 12}, {0x59, 12}, {0x5a, 12}, {0x5b, 12},
281     {0x5c, 12}, {0x5d, 12}, {0x5e, 12}, {0x5f, 12},
282     {0x20, 10}, {0x21, 10}, {0x22, 10}, {0x23, 10},
283     {0x24, 10}, {0x25, 10}, {0x26, 10}, {0x27, 10},
284     {0x28, 10}, {0x29, 10}, {0x2a, 10}, {0x2b, 10},
285     {0x2c, 10}, {0x2d, 10}, {0x2e, 10}, {0x2f, 10},
286     {0x10, 8}, {0x11, 8}, {0x12, 8}, {0x13, 8},
287     {0x14, 8}, {0x15, 8}, {0x16, 8}, {0x17, 8},
288     {0x08, 6}, {0x09, 6}, {0x0a, 6}, {0x0b, 6},
289     {0x04, 4}, {0x05, 4}, {0x04, 3}, {0x03, 2},
290     {0x05, 3}, {0x06, 4}, {0x07, 4}, {0x0c, 6},
```

```
291     {0x0d, 6}, {0x0e, 6}, {0x0f, 6}, {0x18, 8},
292     {0x19, 8}, {0x1a, 8}, {0x1b, 8}, {0x1c, 8},
293     {0x1d, 8}, {0x1e, 8}, {0x1f, 8}, {0x30, 10},
294     {0x31, 10}, {0x32, 10}, {0x33, 10}, {0x34, 10},
295     {0x35, 10}, {0x36, 10}, {0x37, 10}, {0x38, 10},
296     {0x39, 10}, {0x3a, 10}, {0x3b, 10}, {0x3c, 10},
297     {0x3d, 10}, {0x3e, 10}, {0x3f, 10}, {0x60, 12},
298     {0x61, 12}, {0x62, 12}, {0x63, 12}, {0x64, 12},
299     {0x65, 12}, {0x66, 12}, {0x67, 12}, {0x68, 12},
300     {0x69, 12}, {0x6a, 12}, {0x6b, 12}, {0x6c, 12},
301     {0x6d, 12}, {0x6e, 12}, {0x6f, 12}, {0x70, 12},
302     {0x71, 12}, {0x72, 12}, {0x73, 12}, {0x74, 12},
303     {0x75, 12}, {0x76, 12}, {0x77, 12}, {0x78, 12},
304     {0x79, 12}, {0x7a, 12}, {0x7b, 12}, {0x7c, 12},
305     {0x7d, 12}, {0x7e, 12}, {0x7f, 12}, {0xc0, 14},
306     {0xc1, 14}, {0xc2, 14}, {0xc3, 14}, {0xc4, 14},
307     {0xc5, 14}, {0xc6, 14}, {0xc7, 14}, {0xc8, 14},
308     {0xc9, 14}, {0xca, 14}, {0xcb, 14}, {0xcc, 14},
309     {0xcd, 14}, {0xce, 14}, {0xcf, 14}, {0xd0, 14},
310     {0xd1, 14}, {0xd2, 14}, {0xd3, 14}, {0xd4, 14},
311     {0xd5, 14}, {0xd6, 14}, {0xd7, 14}, {0xd8, 14},
312     {0xd9, 14}, {0xda, 14}, {0xdb, 14}, {0xdc, 14},
313     {0xdd, 14}, {0xde, 14}, {0xdf, 14}, {0xe0, 14},
314     {0xe1, 14}, {0xe2, 14}, {0xe3, 14}, {0xe4, 14},
315     {0xe5, 14}, {0xe6, 14}, {0xe7, 14}, {0xe8, 14},
316     {0xe9, 14}, {0xea, 14}, {0xeb, 14}, {0xec, 14},
317     {0xed, 14}, {0xee, 14}, {0xef, 14}, {0xf0, 14},
318     {0xf1, 14}, {0xf2, 14}, {0xf3, 14}, {0xf4, 14},
319     {0xf5, 14}, {0xf6, 14}, {0xf7, 14}, {0xf8, 14},
320     {0xf9, 14}, {0xfa, 14}, {0xfb, 14}, {0xfc, 14},
321     {0xfd, 14}, {0xfe, 14}, {0xff, 14}, {0x180, 16},
322     {0x181, 16}, {0x182, 16}, {0x183, 16}, {0x184, 16},
323     {0x185, 16}, {0x186, 16}, {0x187, 16}, {0x188, 16},
324     {0x189, 16}, {0x18a, 16}, {0x18b, 16}, {0x18c, 16},
325     {0x18d, 16}, {0x18e, 16}, {0x18f, 16}, {0x190, 16},
326     {0x191, 16}, {0x192, 16}, {0x193, 16}, {0x194, 16},
327     {0x195, 16}, {0x196, 16}, {0x197, 16}, {0x198, 16},
328     {0x199, 16}, {0x19a, 16}, {0x19b, 16}, {0x19c, 16},
329     {0x19d, 16}, {0x19e, 16}, {0x19f, 16}, {0x1a0, 16},
330     {0x1a1, 16}, {0x1a2, 16}, {0x1a3, 16}, {0x1a4, 16},
331     {0x1a5, 16}, {0x1a6, 16}, {0x1a7, 16}, {0x1a8, 16},
332     {0x1a9, 16}, {0x1aa, 16}, {0x1ab, 16}, {0x1ac, 16},
333     {0x1ad, 16}, {0x1ae, 16}, {0x1af, 16}, {0x1b0, 16},
334     {0x1b1, 16}, {0x1b2, 16}, {0x1b3, 16}, {0x1b4, 16},
335     {0x1b5, 16}, {0x1b6, 16}, {0x1b7, 16}, {0x1b8, 16},
336     {0x1b9, 16}, {0x1ba, 16}, {0x1bb, 16}, {0x1bc, 16},
```

```
337     {0x1bd, 16}, {0x1be, 16}, {0x1bf, 16}, {0x1c0, 16},
338     {0x1c1, 16}, {0x1c2, 16}, {0x1c3, 16}, {0x1c4, 16},
339     {0x1c5, 16}, {0x1c6, 16}, {0x1c7, 16}, {0x1c8, 16},
340     {0x1c9, 16}, {0x1ca, 16}, {0x1cb, 16}, {0x1cc, 16},
341     {0x1cd, 16}, {0x1ce, 16}, {0x1cf, 16}, {0x1d0, 16},
342     {0x1d1, 16}, {0x1d2, 16}, {0x1d3, 16}, {0x1d4, 16},
343     {0x1d5, 16}, {0x1d6, 16}, {0x1d7, 16}, {0x1d8, 16},
344     {0x1d9, 16}, {0x1da, 16}, {0x1db, 16}, {0x1dc, 16},
345     {0x1dd, 16}, {0x1de, 16}, {0x1df, 16}, {0x1e0, 16},
346     {0x1e1, 16}, {0x1e2, 16}, {0x1e3, 16}, {0x1e4, 16},
347     {0x1e5, 16}, {0x1e6, 16}, {0x1e7, 16}, {0x1e8, 16},
348     {0x1e9, 16}, {0x1ea, 16}, {0x1eb, 16}, {0x1ec, 16},
349     {0x1ed, 16}, {0x1ee, 16}, {0x1ef, 16}, {0x1f0, 16},
350     {0x1f1, 16}, {0x1f2, 16}, {0x1f3, 16}, {0x1f4, 16},
351     {0x1f5, 16}, {0x1f6, 16}, {0x1f7, 16}, {0x1f8, 16},
352     {0x1f9, 16}, {0x1fa, 16}, {0x1fb, 16}, {0x1fc, 16},
353     {0x1fd, 16}, {0x1fe, 16}, {0x1ff, 16},
354 };
```

```
356 VLC_TABLE const coeff_tab[2][102] =
```

```
357 {
358     /* intra = 0 */
359     {
360         {{ 2, 2}, {0, 0, 1}},
361         {{15, 4}, {0, 0, 2}},
362         {{21, 6}, {0, 0, 3}},
363         {{23, 7}, {0, 0, 4}},
364         {{31, 8}, {0, 0, 5}},
365         {{37, 9}, {0, 0, 6}},
366         {{36, 9}, {0, 0, 7}},
367         {{33, 10}, {0, 0, 8}},
368         {{32, 10}, {0, 0, 9}},
369         {{ 7, 11}, {0, 0, 10}},
370         {{ 6, 11}, {0, 0, 11}},
371         {{32, 11}, {0, 0, 12}},
372         {{ 6, 3}, {0, 1, 1}},
373         {{20, 6}, {0, 1, 2}},
374         {{30, 8}, {0, 1, 3}},
375         {{15, 10}, {0, 1, 4}},
376         {{33, 11}, {0, 1, 5}},
377         {{80, 12}, {0, 1, 6}},
378         {{14, 4}, {0, 2, 1}},
379         {{29, 8}, {0, 2, 2}},
380         {{14, 10}, {0, 2, 3}},
381         {{81, 12}, {0, 2, 4}},
382         {{13, 5}, {0, 3, 1}},
```



```
383      {{35, 9}, {0, 3, 2}},
384      {{13, 10}, {0, 3, 3}},
385      {{12, 5}, {0, 4, 1}},
386      {{34, 9}, {0, 4, 2}},
387      {{82, 12}, {0, 4, 3}},
388      {{11, 5}, {0, 5, 1}},
389      {{12, 10}, {0, 5, 2}},
390      {{83, 12}, {0, 5, 3}},
391      {{19, 6}, {0, 6, 1}},
392      {{11, 10}, {0, 6, 2}},
393      {{84, 12}, {0, 6, 3}},
394      {{18, 6}, {0, 7, 1}},
395      {{10, 10}, {0, 7, 2}},
396      {{17, 6}, {0, 8, 1}},
397      {{ 9, 10}, {0, 8, 2}},
398      {{16, 6}, {0, 9, 1}},
399      {{ 8, 10}, {0, 9, 2}},
400      {{22, 7}, {0, 10, 1}},
401      {{85, 12}, {0, 10, 2}},
402      {{21, 7}, {0, 11, 1}},
403      {{20, 7}, {0, 12, 1}},
404      {{28, 8}, {0, 13, 1}},
405      {{27, 8}, {0, 14, 1}},
406      {{33, 9}, {0, 15, 1}},
407      {{32, 9}, {0, 16, 1}},
408      {{31, 9}, {0, 17, 1}},
409      {{30, 9}, {0, 18, 1}},
410      {{29, 9}, {0, 19, 1}},
411      {{28, 9}, {0, 20, 1}},
412      {{27, 9}, {0, 21, 1}},
413      {{26, 9}, {0, 22, 1}},
414      {{34, 11}, {0, 23, 1}},
415      {{35, 11}, {0, 24, 1}},
416      {{86, 12}, {0, 25, 1}},
417      {{87, 12}, {0, 26, 1}},
418      {{ 7, 4}, {1, 0, 1}},
419      {{25, 9}, {1, 0, 2}},
420      {{ 5, 11}, {1, 0, 3}},
421      {{15, 6}, {1, 1, 1}},
422      {{ 4, 11}, {1, 1, 2}},
423      {{14, 6}, {1, 2, 1}},
424      {{13, 6}, {1, 3, 1}},
425      {{12, 6}, {1, 4, 1}},
426      {{19, 7}, {1, 5, 1}},
427      {{18, 7}, {1, 6, 1}},
428      {{17, 7}, {1, 7, 1}},
```

```
429      {{16, 7}, {1, 8, 1}},
430      {{26, 8}, {1, 9, 1}},
431      {{25, 8}, {1, 10, 1}},
432      {{24, 8}, {1, 11, 1}},
433      {{23, 8}, {1, 12, 1}},
434      {{22, 8}, {1, 13, 1}},
435      {{21, 8}, {1, 14, 1}},
436      {{20, 8}, {1, 15, 1}},
437      {{19, 8}, {1, 16, 1}},
438      {{24, 9}, {1, 17, 1}},
439      {{23, 9}, {1, 18, 1}},
440      {{22, 9}, {1, 19, 1}},
441      {{21, 9}, {1, 20, 1}},
442      {{20, 9}, {1, 21, 1}},
443      {{19, 9}, {1, 22, 1}},
444      {{18, 9}, {1, 23, 1}},
445      {{17, 9}, {1, 24, 1}},
446      {{ 7, 10}, {1, 25, 1}},
447      {{ 6, 10}, {1, 26, 1}},
448      {{ 5, 10}, {1, 27, 1}},
449      {{ 4, 10}, {1, 28, 1}},
450      {{36, 11}, {1, 29, 1}},
451      {{37, 11}, {1, 30, 1}},
452      {{38, 11}, {1, 31, 1}},
453      {{39, 11}, {1, 32, 1}},
454      {{88, 12}, {1, 33, 1}},
455      {{89, 12}, {1, 34, 1}},
456      {{90, 12}, {1, 35, 1}},
457      {{91, 12}, {1, 36, 1}},
458      {{92, 12}, {1, 37, 1}},
459      {{93, 12}, {1, 38, 1}},
460      {{94, 12}, {1, 39, 1}},
461      {{95, 12}, {1, 40, 1}}
462  },
463  /* intra = 1 */
464  {
465      {{ 2, 2}, {0, 0, 1}},
466      {{15, 4}, {0, 0, 3}},
467      {{21, 6}, {0, 0, 6}},
468      {{23, 7}, {0, 0, 9}},
469      {{31, 8}, {0, 0, 10}},
470      {{37, 9}, {0, 0, 13}},
471      {{36, 9}, {0, 0, 14}},
472      {{33, 10}, {0, 0, 17}},
473      {{32, 10}, {0, 0, 18}},
474      {{ 7, 11}, {0, 0, 21}},
```

```
475      {{ 6, 11}, {0, 0, 22}},
476      {{32, 11}, {0, 0, 23}},
477      {{ 6,  3}, {0, 0, 2}},
478      {{20,  6}, {0, 1, 2}},
479      {{30,  8}, {0, 0, 11}},
480      {{15, 10}, {0, 0, 19}},
481      {{33, 11}, {0, 0, 24}},
482      {{80, 12}, {0, 0, 25}},
483      {{14,  4}, {0, 1, 1}},
484      {{29,  8}, {0, 0, 12}},
485      {{14, 10}, {0, 0, 20}},
486      {{81, 12}, {0, 0, 26}},
487      {{13,  5}, {0, 0, 4}},
488      {{35,  9}, {0, 0, 15}},
489      {{13, 10}, {0, 1, 7}},
490      {{12,  5}, {0, 0, 5}},
491      {{34,  9}, {0, 4, 2}},
492      {{82, 12}, {0, 0, 27}},
493      {{11,  5}, {0, 2, 1}},
494      {{12, 10}, {0, 2, 4}},
495      {{83, 12}, {0, 1, 9}},
496      {{19,  6}, {0, 0, 7}},
497      {{11, 10}, {0, 3, 4}},
498      {{84, 12}, {0, 6, 3}},
499      {{18,  6}, {0, 0, 8}},
500      {{10, 10}, {0, 4, 3}},
501      {{17,  6}, {0, 3, 1}},
502      {{ 9, 10}, {0, 8, 2}},
503      {{16,  6}, {0, 4, 1}},
504      {{ 8, 10}, {0, 5, 3}},
505      {{22,  7}, {0, 1, 3}},
506      {{85, 12}, {0, 1, 10}},
507      {{21,  7}, {0, 2, 2}},
508      {{20,  7}, {0, 7, 1}},
509      {{28,  8}, {0, 1, 4}},
510      {{27,  8}, {0, 3, 2}},
511      {{33,  9}, {0, 0, 16}},
512      {{32,  9}, {0, 1, 5}},
513      {{31,  9}, {0, 1, 6}},
514      {{30,  9}, {0, 2, 3}},
515      {{29,  9}, {0, 3, 3}},
516      {{28,  9}, {0, 5, 2}},
517      {{27,  9}, {0, 6, 2}},
518      {{26,  9}, {0, 7, 2}},
519      {{34, 11}, {0, 1, 8}},
520      {{35, 11}, {0, 9, 2}},
```

```
521      {{86, 12}, {0, 2, 5}},
522      {{87, 12}, {0, 7, 3}},
523      {{ 7,  4}, {1, 0, 1}},
524      {{25,  9}, {0, 11, 1}},
525      {{ 5, 11}, {1, 0, 6}},
526      {{15,  6}, {1, 1, 1}},
527      {{ 4, 11}, {1, 0, 7}},
528      {{14,  6}, {1, 2, 1}},
529      {{13,  6}, {0, 5, 1}},
530      {{12,  6}, {1, 0, 2}},
531      {{19,  7}, {1, 5, 1}},
532      {{18,  7}, {0, 6, 1}},
533      {{17,  7}, {1, 3, 1}},
534      {{16,  7}, {1, 4, 1}},
535      {{26,  8}, {1, 9, 1}},
536      {{25,  8}, {0, 8, 1}},
537      {{24,  8}, {0, 9, 1}},
538      {{23,  8}, {0, 10, 1}},
539      {{22,  8}, {1, 0, 3}},
540      {{21,  8}, {1, 6, 1}},
541      {{20,  8}, {1, 7, 1}},
542      {{19,  8}, {1, 8, 1}},
543      {{24,  9}, {0, 12, 1}},
544      {{23,  9}, {1, 0, 4}},
545      {{22,  9}, {1, 1, 2}},
546      {{21,  9}, {1, 10, 1}},
547      {{20,  9}, {1, 11, 1}},
548      {{19,  9}, {1, 12, 1}},
549      {{18,  9}, {1, 13, 1}},
550      {{17,  9}, {1, 14, 1}},
551      {{ 7, 10}, {0, 13, 1}},
552      {{ 6, 10}, {1, 0, 5}},
553      {{ 5, 10}, {1, 1, 3}},
554      {{ 4, 10}, {1, 2, 2}},
555      {{36, 11}, {1, 3, 2}},
556      {{37, 11}, {1, 4, 2}},
557      {{38, 11}, {1, 15, 1}},
558      {{39, 11}, {1, 16, 1}},
559      {{88, 12}, {0, 14, 1}},
560      {{89, 12}, {1, 0, 8}},
561      {{90, 12}, {1, 5, 2}},
562      {{91, 12}, {1, 6, 2}},
563      {{92, 12}, {1, 17, 1}},
564      {{93, 12}, {1, 18, 1}},
565      {{94, 12}, {1, 19, 1}},
566      {{95, 12}, {1, 20, 1}}
```

```
567     }
568 };

570 uint8_t const max_level[2][2][64] = {
571     {
572         /* intra = 0, last = 0 */
573         {
574             12, 6, 4, 3, 3, 3, 3, 2,
575             2, 2, 2, 1, 1, 1, 1, 1,
576             1, 1, 1, 1, 1, 1, 1, 1,
577             1, 1, 1, 0, 0, 0, 0, 0,
578             0, 0, 0, 0, 0, 0, 0, 0,
579             0, 0, 0, 0, 0, 0, 0, 0,
580             0, 0, 0, 0, 0, 0, 0, 0,
581             0, 0, 0, 0, 0, 0, 0, 0
582         },
583         /* intra = 0, last = 1 */
584         {
585             3, 2, 1, 1, 1, 1, 1, 1,
586             1, 1, 1, 1, 1, 1, 1, 1,
587             1, 1, 1, 1, 1, 1, 1, 1,
588             1, 1, 1, 1, 1, 1, 1, 1,
589             1, 1, 1, 1, 1, 1, 1, 1,
590             1, 0, 0, 0, 0, 0, 0, 0,
591             0, 0, 0, 0, 0, 0, 0, 0,
592             0, 0, 0, 0, 0, 0, 0, 0
593         }
594     },
595     {
596         /* intra = 1, last = 0 */
597         {
598             27, 10, 5, 4, 3, 3, 3, 3,
599             2, 2, 1, 1, 1, 1, 1, 0,
600             0, 0, 0, 0, 0, 0, 0, 0,
601             0, 0, 0, 0, 0, 0, 0, 0,
602             0, 0, 0, 0, 0, 0, 0, 0,
603             0, 0, 0, 0, 0, 0, 0, 0,
604             0, 0, 0, 0, 0, 0, 0, 0,
605             0, 0, 0, 0, 0, 0, 0, 0
606         },
607         /* intra = 1, last = 1 */
608         {
609             8, 3, 2, 2, 2, 2, 2, 1,
610             1, 1, 1, 1, 1, 1, 1, 1,
611             1, 1, 1, 1, 1, 0, 0, 0,
612             0, 0, 0, 0, 0, 0, 0, 0,
```

```
613         0, 0, 0, 0, 0, 0, 0, 0, 0,
614         0, 0, 0, 0, 0, 0, 0, 0, 0,
615         0, 0, 0, 0, 0, 0, 0, 0, 0,
616         0, 0, 0, 0, 0, 0, 0, 0, 0
617     }
618 }
619 };

621 uint8_t const max_run[2][2][64] = {
622     {
623         /* intra = 0, last = 0 */
624         {
625             0, 26, 10, 6, 2, 1, 1, 0,
626             0, 0, 0, 0, 0, 0, 0, 0,
627             0, 0, 0, 0, 0, 0, 0, 0,
628             0, 0, 0, 0, 0, 0, 0, 0,
629             0, 0, 0, 0, 0, 0, 0, 0,
630             0, 0, 0, 0, 0, 0, 0, 0,
631             0, 0, 0, 0, 0, 0, 0, 0,
632             0, 0, 0, 0, 0, 0, 0, 0,
633         },
634         /* intra = 0, last = 1 */
635         {
636             0, 40, 1, 0, 0, 0, 0, 0,
637             0, 0, 0, 0, 0, 0, 0, 0,
638             0, 0, 0, 0, 0, 0, 0, 0,
639             0, 0, 0, 0, 0, 0, 0, 0,
640             0, 0, 0, 0, 0, 0, 0, 0,
641             0, 0, 0, 0, 0, 0, 0, 0,
642             0, 0, 0, 0, 0, 0, 0, 0,
643             0, 0, 0, 0, 0, 0, 0, 0,
644         }
645     },
646     {
647         /* intra = 1, last = 0 */
648         {
649             0, 14, 9, 7, 3, 2, 1, 1,
650             1, 1, 1, 0, 0, 0, 0, 0,
651             0, 0, 0, 0, 0, 0, 0, 0,
652             0, 0, 0, 0, 0, 0, 0, 0,
653             0, 0, 0, 0, 0, 0, 0, 0,
654             0, 0, 0, 0, 0, 0, 0, 0,
655             0, 0, 0, 0, 0, 0, 0, 0,
656             0, 0, 0, 0, 0, 0, 0, 0,
657         },
658         /* intra = 1, last = 1 */
```

```
659     {
660         0, 20, 6, 1, 0, 0, 0, 0,
661         0, 0, 0, 0, 0, 0, 0, 0,
662         0, 0, 0, 0, 0, 0, 0, 0,
663         0, 0, 0, 0, 0, 0, 0, 0,
664         0, 0, 0, 0, 0, 0, 0, 0,
665         0, 0, 0, 0, 0, 0, 0, 0,
666         0, 0, 0, 0, 0, 0, 0, 0,
667         0, 0, 0, 0, 0, 0, 0, 0,
668     }
669 }
670 };
```

```
673 #endif
```

8.5 mbcoding.h 文件

8.5.1 功能描述

初始化 VLC 表数组的函数原型声明, Intra 宏块和 Inter 宏块编码函数的原型声明。

8.5.2 文件注释

```
1  #ifndef _MB_CODING_H_
2  #define _MB_CODING_H_

4  void init_vlc_tables(void);

6  void CodeBlockIntra(const FRAMEINFO* const frame, const MACROBLOCK* pMB,
7                      int16_t qcoeff[6 * 64], Bitstream * bs);

9  void CodeBlockInter(const FRAMEINFO* const frame, const MACROBLOCK* pMB,
10                     int16_t qcoeff[6 * 64], Bitstream * bs);

12 void CodeBlockInters(const FRAMEINFO* const frame, const MACROBLOCK* pMB,
13                     Bitstream * bs);

15 #endif
```

8.6 mbcoding.c 文件

8.6.1 功能描述

初始化 VLC 表数组的函数实现，Intra 宏块和 Inter 宏块编码函数及其辅助函数的实现。

/*

运动矢量编码举例，使用 13818-2 mvd 表

假设一个子图具有下面的运动矢量值：**3 10 30 30 -14 -16 27 24**

矢量编码的模取 2 适合范围的要求，初始化预测值为 0，于是差分为：**7 20 0 -44 -2 43 -3**

对应于 f=2，加上或减去模 64 可把差分值调整到 [-32, 31] 范围内：**7 20 0 20 -2 -21 -3**

为建立码字， $(mvd + (\text{sign}(mvd) * (f-1)))$ 被 f 除，利用带符号的商从 mvd 表中查找一个变长码字，余数的绝对值用于产生附于变长码后的定长码。

令： $V = (mvd + (\text{sign}(mvd) * (f-1)))$ ； $W = V/f$ ； $R = V \% f$

本例产生的码如下所示：

Value	V	W (f=2)		R	VLC code (W + R)
		idiv	vlc code		
3	4	2	0010	0	0010 0
7	8	4	0000 110	0	0000 1100
20	21	10	0000 0100 10	1	0000 0100 101
0	0	0	1	0	1
20	21	10	0000 0100 10	1	0000 0100 101
-2	-3	-1	011	-1	0111
-21	-22	-11	0000 0100 011	0	0000 0100 0110
-3	-4	-2	0011	0	0011 0

MVD VLC Table (13818-2)

motion	code	motion	code
0000 0011 001	-16	0000 0011 000	16
0000 0011 011	-15	0000 0011 010	15
0000 0011 101	-14	0000 0011 100	14
0000 0011 111	-13	0000 0011 110	13
0000 0100 001	-12	0000 0100 000	12
0000 0100 011	-11	0000 0100 010	11
0000 0100 11	-10	0000 0100 10	10
0000 0101 01	-9	0000 0101 00	9
0000 0101 11	-8	0000 0101 10	8
0000 0111	-7	0000 0110	7
0000 1001	-6	0000 1000	6
0000 1011	-5	0000 1010	5
0000 111	-4	0000 110	4
0001 1	-3	0001 0	3
0011	-2	0010	2
011	-1	010	1
1	0	1	0

// */

8.6.2 文件注释

```
1  #include <math.h>

3  #include "../portab.h"
4  #include "../global.h"
5  #include "../encoder.h"

7  #include "bitstream.h"
8  #include "vlc_codes.h"
9  #include "mbcoding.h"

11 #define LEVELOFFSET 32

13 static VLC coeff_VLC[2][2][64][64];
/* 第 13 行定义 coeff_VLC 四维数组, 各维依次是 intra/inter 指示, last or not 指示, level 大小,
   run 长度。注意对 intra 宏块而言, level 大小为[0, 63], 对 inter 宏块而言, level 大小为[-32, 31],
   因此对 inter 有很多特别代码来处理 level 为负的情况, 并且有越界的情况。Intra 宏块和 inter
   宏块的 run 取值范围是[0, 63], 不会越界。
// */

15 void init_vlc_tables(void)
16 {
17     uint32_t i, offset;
18     uint32_t intra, last, run, run_esc, level, level_esc, escape, escape_len;

20     uint32_t last0, run0, level0, code0, len0;

22     for (intra = 0; intra < 2; intra++)
23     {
24         offset = !intra * LEVELOFFSET;

26         for (last = 0; last < 2; last++)
27         {
28             for (run = 0; run < 63 + last; run++)
29             {
30                 for (level = 0; level < (uint32_t)(32 << intra); level++)
31                 {
32                     coeff_VLC[intra][last][level + offset][run].len = 128;
33                 }
```

```
34         }
35     }
36 }
/* 第 22 行到第 36 行初始化 len 变量为 128，指示对应的 code 无效。
// */
38     for (intra = 0; intra < 2; intra++)
39     {
40         offset = !intra * LEVELOFFSET;

42         for (i = 0; i < 102; i++)
43         {
44             last0 = coeff_tab[intra][i].event.last;
45             level0 = coeff_tab[intra][i].event.level + offset;
46             run0 = coeff_tab[intra][i].event.run;

48             coeff_VLC[intra][last0][level0][run0].code
49                 = coeff_tab[intra][i].vlc.code << 1;

51             coeff_VLC[intra][last0][level0][run0].len
52                 = coeff_tab[intra][i].vlc.len + 1;

54             if (!intra)
55             {
56                 level0 = offset - coeff_tab[intra][i].event.level;

58                 coeff_VLC[intra][last0][level0][run0].code
59                     = (coeff_tab[intra][i].vlc.code << 1) | 1;
60                 coeff_VLC[intra][last0][level0][run0].len
61                     = coeff_tab[intra][i].vlc.len + 1;
62             }
63         }
64     }
/* 第 38 行到第 64 行用 coeff _tab 初始化 coeff_VLC 数组，其中第 48 行到第 49 行生成当 level 值
   为正数时的 VLC 表值，把 code 值简单右移一位就是在后面拼接的 0 值指示正数，其中第 54 行到
   第 62 行生成 inter level 为负数时的 VLC 表值，先把 code 值右移一位，拼上 1 指示负数再赋值
   给 code 值，长度加 1 后赋值给 len。
// */
66     for (intra = 0; intra < 2; intra++)
67     {
68         offset = !intra * LEVELOFFSET;
```

```
70         for (last = 0; last < 2; last++)
71         {
72             for (run = 0; run < 63 + last; run++)
73             {
74                 for (level = 1; level < (uint32_t)(32 << intra); level++)
75                 {
76                     if (level <= max_level[intra][last][run]
77                         && run <= max_run[intra][last][level])
78                         continue;
79 /* 第 76 行到第 78 行处理已经初始化过的数组项，就直接进入下一轮循环。
80 // */
81
82                     level_esc = level - max_level[intra][last][run];
83                     run_esc = run - 1 - max_run[intra][last][level];
84
85                     if (level_esc <= max_level[intra][last][run]
86                         && run_esc <= max_run[intra][last][level_esc])
87                     {
88                         escape      = ESCAPE1;
89                         escape_len = 7 + 1;
90                         run_esc     = run;
91 /* 第 86 行到第 88 行计算有些不能直接通过 intra TCOEFF 和 inter TCOEFF 表来编码，但是通过将当前
92 level 值减去当前 run 对应的 LMAX 后，就可以用 TCOEFF 表来编码的那些码字的 VLC 参数。
93 // */
94
95                     }
96                     else
97                     {
98                         if (run_esc <= max_run[intra][last][level]
99                             && level <= max_level[intra][last][run_esc])
100                         {
101                             escape      = ESCAPE2;
102                             escape_len = 7 + 2;
103                             level_esc  = level;
104 /* 第 95 行到第 97 行计算有些不能直接通过 intra TCOEFF 和 inter TCOEFF 表来编码，但是通过将当前
105 run 值减去当前 level 对应的(RMAX+1)后，就可以用 TCOEFF 表来编码的那些码字的 VLC 参数。
106 // */
107
108                         }
109                         else if (!intra)
110                         {
```

```

101             code0=(ESCAPE3<<21)|(last<<20)|(run<<14)|(1<<13);

103             coeff_VLC[intra][last][level + offset][run].code
104                 = code0 | ((level&0xfff)<<1)|1;
105             coeff_VLC[intra][last][level + offset][run].len = 30;

107             coeff_VLC[intra][last][offset - level][run].code
108                 = code0 | ((-(int32_t)level&0xfff)<<1)|1;
109             coeff_VLC[intra][last][offset - level][run].len = 30;

111             continue;
/* 第 101 行到第 111 行生成 ESC 模式三定长编码的 VLC 码表项，这种情况下分别用 1 比特编 LAST，6 比
特编 RUN 和 12 比特编 LEVEL。为了避免和 resync_marker 相同，分别有一个标记比特插在 12 比特的 LEVEL
的前面和后面。
// */
112             }
113             else
114             {
115                 continue;
116             }
117         }

119         level0 = level + offset;

121         len0 = coeff_VLC[intra][last][level_esc + offset][run_esc].len;
122         code0 =coeff_VLC[intra][last][level_esc + offset][run_esc].code;

124         coeff_VLC[intra][last][level0][run].code
125             = (escape << len0) | code0;
126         coeff_VLC[intra][last][level0][run].len
127             = len0 + escape_len;
/* 第 119 行到第 127 行依照第 86 行到第 88 行和第 95 行到第 97 行计算的 VLC 参数，生成 ESC 编码模
式一和编码模式二相应的 VLC 表项。
// */
129             if (!intra)
130             {
131                 coeff_VLC[intra][last][offset - level][run].code
132                     = (escape << len0) | code0 | 1;
133                 coeff_VLC[intra][last][offset - level][run].len
134                     = len0 + escape_len;

```

```
135             }
/* 第 129 行到第 135 行依照第 86 行到第 88 行和第 95 行到第 97 行计算的 VLC 参数，生成 inter level
为负数时 ESC 编码模式一和编码模式二相应的 VLC 表项。
// */
136         }

138         if (!intra)
139         {
140             code0 = (ESCAPE3<<21) | (last<<20) | (run<<14) | (1<<13);

142             coeff_VLC[intra][last][0][run].code
143                 = code0 | ((-32 & 0xfff) << 1) | 1;
144             coeff_VLC[intra][last][0][run].len = 30;
145         }
/* 由第 74 行代码可知每次循环只处理[0, 31]区间的正负值，所以要第 138 行到第 145 行代码来处理
inter level 为-32 时的 VLC 表项。
// */
146     }
147 }
148 }
149 }

151 static __inline void CodeVector(Bitstream * bs, int32_t value, int32_t f_code)
152 { // 先理解前面那个运动矢量举例。
153     if (value == 0)
154     {
155         BitstreamPutBit1(bs);
156     }
157     else
158     {
159         uint16_t length, code, mv_res, sign;

161         length = 16 << f_code;
162         f_code--;

164         sign = (value < 0);

166         if (value >= length)
167             value -= 2 * length;
168         else if (value < -length)
```

```
169         value += 2 * length;
/* 第 166 行到第 169 行用矢量的模来限定差分矢量的大小，避免矢量越界。
// */

171         if (sign)
172             value = -value;

174         value--;
175         mv_res = value & ((1 << f_code) - 1);    // 求商
176         code = ((value - mv_res) >> f_code) + 1; // 求余数

178         if (sign)
179             code = -code;

181         code += 32; // 编码有符号商
182         BitstreamPutBits(bs, mb_motion_table[code].code, mb_motion_table[code].len);

184         if (f_code) // 编码余数
185             BitstreamPutBits(bs, mv_res, f_code);
186     }
187 }

189 static __inline void CodeCoeffInter(Bitstream* bs, const int16_t qcoeff[64])
190 {
191     uint32_t i, run, prev_run, code, len;
192     int32_t level, prev_level, level_shifted;

194     i = 0;
195     run = 0;

197     while (!(level = qcoeff[zigzag[i++]])) // 找第一个非零系数
198         run++;

200     prev_level = level; // 因为现在不知道是否是最后一个非零系数，所以保存 level 和 run 值
201     prev_run = run;
202     run = 0;

204     while (i < 64)
205     {
206         if ((level = qcoeff[zigzag[i++]]) != 0)
```

```
207     {
208         level_shifted = prev_level + 32;
209         if (!(level_shifted & -64))
210         {
211             code = coeff_VLC[0][0][level_shifted][prev_run].code;
212             len = coeff_VLC[0][0][level_shifted][prev_run].len;
213         }
214         else
215         {
216             code=(ESCAPE3<<21) | (prev_run<<14) | (1<<13) | ((prev_level&0xffff)<<1) | 1;
217             len =30;
218         }
219         BitstreamPutBits(bs, code, len);
220     /* 第 208 行到第 219 行编码一个非 last (level,run) 对, 用 level 是否超 [-32, 31] 来判断是否在
221        coeff_VLC 数组中, run 的取值范围不会越界, 不在 coeff_VLC 数组中时用 ESC 模式三编码。
222     */
223     prev_level = level;
224     prev_run = run;
225     run = 0;
226 }

228 level_shifted = prev_level + 32;
229 if (!(level_shifted & -64))
230 {
231     code = coeff_VLC[0][1][level_shifted][prev_run].code;
232     len = coeff_VLC[0][1][level_shifted][prev_run].len;
233 }
234 else
235 {
236     code=(ESCAPE3<<21) | (1<<20) | (prev_run<<14) | (1<<13) | ((prev_level&0xffff)<<1) | 1;
237     len = 30;
238 }
239 BitstreamPutBits(bs, code, len);
240 /* 第 228 行到第 239 行编码一个 last (level,run) 对, 用 level 是否超 [-32, 31] 来判断是否在 coeff_VLC
241    数组中, run 的取值范围是 [0, 63] 不会越界, 不在 coeff_VLC 数组中时用 ESC 模式三编码。
242 */
243 */
244 }
```

```
242 static __inline void CodeCoeffIntra(Bitstream* bs, const int16_t qcoeff[64])
243 {
244     uint32_t i, abs_level, run, prev_run, code, len;
245     int32_t level, prev_level;

246
247     i = 1;
248     run = 0;

249
250     while (i<64 && !(level = qcoeff[zigzag[i++]])) // 找第一个非零系数
251         run++;

252
253     prev_level = level; // 因为现在不知道是否是最后一个非零系数，所以保存 level 和 run 值
254     prev_run = run;
255     run = 0;

256
257     while (i < 64)
258     {
259         if ((level = qcoeff[zigzag[i++]]) != 0)
260         {
261             abs_level = abs(prev_level);
262             abs_level = abs_level < 64 ? abs_level : 0;
263             code = coeff_VLC[1][0][abs_level][prev_run].code;
264             len = coeff_VLC[1][0][abs_level][prev_run].len;
265             if (len != 128)
266                 code |= (prev_level < 0);
267             else
268             {
269                 code=(ESCAPE3<<21)|(prev_run<<14)|(1<<13)|((prev_level&0xffff)<<1)|1;
270                 len = 30;
271             }
272             BitstreamPutBits(bs, code, len);
273             /* 第 263 行到第 274 行编码一个非 last (level, run)对，用 len 是否等于 128 来判断是否在 coeff_VLC
                数组中，不在 coeff_VLC 数组中时用 ESC 模式三编码。
                // */
274             prev_level = level;
275             prev_run = run;
276             run = 0;
277         }
278         else
279         {
```



```
278         run++;
279     }

281     abs_level = abs(prev_level);
282     abs_level = abs_level < 64 ? abs_level : 0;
283     code      = coeff_VLC[1][1][abs_level][prev_run].code;
284     len       = coeff_VLC[1][1][abs_level][prev_run].len;
285     if (len != 128)
286         code |= (prev_level < 0);
287     else
288     {
289         code=(ESCAPE3<<21) | (1<<20) | (prev_run<<14) | (1<<13) | ((prev_level&0xfff)<<1) | 1;
290         len = 30;
291     }
292     BitstreamPutBits(bs, code, len);
/* 第 281 行到第 292 行编码一个 last (level, run)对, 用len 是否等于 128 来判断是否在 coeff_VLC 数
   组中, 不在 coeff_VLC 数组中时用 ESC 模式三编码。
// */
293 }

295 void CodeBlockIntra(const FRAMEINFO* const frame, const MACROBLOCK* pMB,
296                   int16_t qcoeff[6 * 64], Bitstream* bs)
297 {
298     uint32_t i, mcbpc, cbpy, bits;

300     cbpy = pMB->cbp >> 2;

302     if (frame->coding_type == I_VOP)
303     {
304         mcbpc = ((pMB->mode >> 1) & 3) | ((pMB->cbp & 3) << 2);
305         BitstreamPutBits(bs, mcbpc_intra_tab[mcbpc].code, mcbpc_intra_tab[mcbpc].len);
306     }
307     else
308     {
309         mcbpc = (pMB->mode & 7) | ((pMB->cbp & 3) << 3);
310         BitstreamPutBits(bs, mcbpc_inter_tab[mcbpc].code, mcbpc_inter_tab[mcbpc].len);
311     }

313     BitstreamPutBits(bs, 0, 1);
```

```
315     BitstreamPutBits(bs, xvid_cbpy_tab[cbpy].code, xvid_cbpy_tab[cbpy].len);

317     for (i = 0; i < 6; i++)
318     {
319         if (i < 4)
320             BitstreamPutBits(bs, dcy_tab[qcoeff[i * 64] + 255].code,
321                             dcy_tab[qcoeff[i * 64] + 255].len);
322         else
323             BitstreamPutBits(bs, dcc_tab[qcoeff[i * 64] + 255].code,
324                             dcc_tab[qcoeff[i * 64] + 255].len);

326         if (pMB->cbp & (1 << (5 - i)))
327         {
328             bits = BitstreamPos(bs);

330             CodeCoeffIntra(bs, &qcoeff[i * 64]);

332             bits = BitstreamPos(bs) - bits;
333         }
334     }
335 }

/* 第 295 行到第 335 行编码一个 intra 宏块, 先写 cbpc, 写没有 AC 高级预测标准位, 写 cbpy, 写 DC
系数, 再写 AC 系数。” MP4 标准的流水帐”。
// */
```

```
337 void CodeBlockInter(const FRAMEINFO* const frame, const MACROBLOCK* pMB,
338                     int16_t qcoeff[6 * 64], Bitstream* bs)
339 {
340     int32_t i;

342     uint32_t mcbpc = (pMB->mode & 7) | ((pMB->cbp & 3) << 3);
343     uint32_t cbpy = 15 - (pMB->cbp >> 2);

345     BitstreamPutBits(bs, mcbpc_inter_tab[mcbpc].code, mcbpc_inter_tab[mcbpc].len);

347     BitstreamPutBits(bs, xvid_cbpy_tab[cbpy].code, xvid_cbpy_tab[cbpy].len);

349     CodeVector(bs, pMB->pmvs.x, frame->fcode);
350     CodeVector(bs, pMB->pmvs.y, frame->fcode);
```

```

352     for (i = 0; i < 6; i++)
353         if (pMB->cbp & (1 << (5 - i)))
354         {
355             CodeCoeffInter(bs, &qcoeff[i * 64]);
356         }
357 }
/* 第 337 行到第 357 行编码一个 inter 宏块, 先写 cbpc, 写 cbpy, 写运动矢量, 再写 AC 系数。
// */

359 void CodeBlockInters(const FRAMEINFO* const frame, const MACROBLOCK* pMB,
360                     Bitstream* bs)
361 {
362     uint32_t mcbpc = (pMB->mode & 7) | ((pMB->cbp & 3) << 3);
363     uint32_t cbpy = 15 - (pMB->cbp >> 2);

365     BitstreamPutBits(bs, mcbpc_inter_tab[mcbpc].code, mcbpc_inter_tab[mcbpc].len);

367     BitstreamPutBits(bs, xvid_cbpy_tab[cbpy].code, xvid_cbpy_tab[cbpy].len);

369     CodeVector(bs, pMB->pmvs.x, frame->fcode);
370     CodeVector(bs, pMB->pmvs.y, frame->fcode);
371 }
/* 第 359 行到第 371 行编码一个 inter 跳跃宏块, 先写 cbpc, 写 cbpy。
// */

```

8.7 bitstream.h 文件

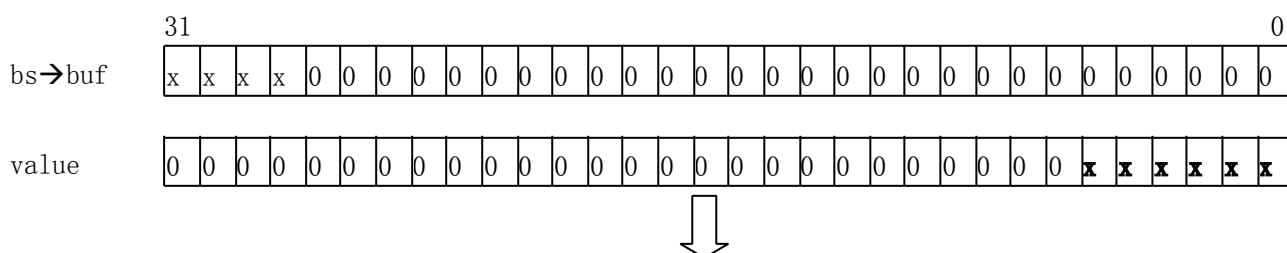
8.7.1 功能描述

主要是往码流中写 bit 位的操作函数的实现, 附带声明写 VOL 头和 VOP 头两函数原型。为了减少 if 判断带来的 CPU 指令流水线中断, 主要优化了 BitstreamPutBits() 通用函数, 并派生出 BitstreamPutBit0() 和 BitstreamPutBit1() 两特殊函数, 最后删除了解码相关的源代码, 所以整个文件代码比较简洁。

/*

码流操作举例(没越界)

```
BitstreamPutBits(bs, value, size); // bs->pos=4; size=6
```



shift=32 - bs→pos - size

[illegible][illegible]

bs→pos=10

码流操作举例(越界)

```
BitstreamPutBits(bs, value, size); // bs->pos=30;size=6
```

value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x5	x4	x3	x2	x1	x0
-------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----



```
shift= bs→pos + size-32=4
```

[illegible]

```
*bs->tail++ = BSWAP(buf | (value>>shift))
```

[illegible]
$$\text{shift} = 64 - \text{bs} \rightarrow \text{pos} - \text{size} = 28$$
[illegible][illegible]

bs→pos=4

 $// \quad *$

8.7.2 文件注释

```
1  #ifndef  BITSTREAM_H
```

```
2  #define  BITSTREAM H
```

```
4 void BitstreamWriteVolHeader(Bitstream * const bs, const MBParam * pParam,
5                             const FRAMEINFO * const frame);
```

```
7 void BitstreamWriteVopHeader(Bitstream * const bs, const MBParam * pParam,  
8                             const FRAMEINFO * const frame, unsigned int quant):
```

```
10 #define WRITE_MARKER() BitstreamPutBit1(bs)
```

/* 第 10 行宏定义, 简单的在码流中写上一位的数字 1 做为一个标记(marker)。

// */

```
/*
```

码流操作提要:

1: 程序开辟一个大缓冲区, 每次往一个临时变量尾巴上拼接 bit 位, 每次拼接后都要判断是否拼接到 32bits, 当拼接到 32bits 时就调整一下大端和小端的差别, 把这一个大 WORD 写到大缓冲区中,

临时变量清零重新拼接。因此需要一个临时变量来保存拼接的 bit 位的值(程序中是 bs->buf), 另一个变量来指示拼接了多少位(程序中是 bs->pos), 缓冲区首地址(程序中是 bs->start)。

2: 每次拼接都是左右移位需要拼接的数到 32bits 位中的相应位置, 已拼接好的数(bs->buf)不需要移位。

```
// */
```

```
12 static __inline void BitstreamInit(Bitstream * const bs, void *const bitstream,
13                                     uint32_t length)
14 {
15     bs->buf = 0;
16     bs->pos = 0;
17     bs->start = bs->tail = (uint32_t *) bitstream;
18 }
```

/* 第 15 行到第 21 行码流初始化, 第 18 行设置码流中 bit 位拼接值为零, 第 19 行拼接的 bit 位为零, 第 20 行设置码流的起始地址和结尾地址为输出缓冲区首地址

```
// */
```

```
20 static __inline void BitstreamPutBit0(Bitstream * const bs)
21 {
22     bs->pos ++;

24     if (bs->pos >= 32)
25     {
26         uint32_t b = bs->buf;
27         BSWAP(b);
28         *bs->tail++ = b;

30         bs->buf = 0;
31         bs->pos = 0;
32     }
33 }
/*
```

第 23 行到第 36 行是往码流中写 bit 位为 1, 值为 0 的数。

第 25 行把位置指针加 1, 因为 bs->buf 初值为 0, 写 0 就不需要改 bs->buf 的值。

第 27 行判断是否拼接到 32bits, 如果是就把这个大 WORD 值写到缓冲区,

码流字节指针加 1, 拼接的 bit 位值置 0, 拼接的 bit 数置 0。

// */

```
35 static __inline void BitstreamPutBit1(Bitstream * const bs)
36 {
37     bs->buf |= (0x80000000 >> bs->pos);

39     bs->pos ++;

41     if (bs->pos >= 32)
42     {
43         uint32_t b = bs->buf;
44         BSWAP(b);
45         *bs->tail++ = b;

47         bs->buf = 0;
48         bs->pos = 0;
49     }
50 }
/*
```

第 38 行到第 53 行是往码流中写 bit 位为 1, 值为 1 的数。

第 40 行先把 1bit 数字 1 偏移到适当位置再添加到已拼接值(bs->buf)中

第 42 行把位置指针加 1。

第 44 行判断是否拼接到 32bits, 如果是就把这个 DWORD 值写到缓冲区,

码流字节指针加 1, 拼接的 bit 位值置 0, 拼接的 bit 数置 0。

// */

```
52 static __inline void BitstreamPutBits(Bitstream * const bs, const uint32_t value,
53     const uint32_t size)
54 {
55     int32_t shift = 32 - bs->pos - size;

57     if (shift > 0)
58     {
59         bs->buf |= value << shift;
60         bs->pos += size;
61     }
62     else
63     {
64         shift = bs->pos + size - 32;
65         bs->buf |= value >> shift;
```

```
67     {
68         uint32_t b = bs->buf;
69         BSWAP(b);
70         *bs->tail++ = b;
71     }

73     bs->pos += size - 32;
74     if(bs->pos) bs->buf = value << (32 - bs->pos);
75     else      bs->buf = 0;
76 }
77 }
/*
```

第 55 行到第 80 行是往码流中拼接值为 value, bit 位数为 size 的 bit 串。

第 58 行计算如果完全拼接上越界值。

第 60 行是判断如果全部拼接上是否越界(大于等于 32bit)。

第 62 行到第 63 行处理没有越界的情况, 直接往上拼接就行, 修改相应变量值。

第 67 行到第 74 行取部分需拼接的数(value)拼接完整一个 DWORD 值并写到码流中。

第 76 行保存还剩下多少个 bit 位。

第 77 行保存大于 32bits 位时剩下那些 bit 位表示的值。

第 78 行设置刚好等于 32bits 位时值为 0。此时 size 位全部用完, 没有剩余 bit 位。

```
// */
```

```
79 static __inline uint32_t BitstreamPos(const Bitstream * const bs)
80 {
81     return((uint32_t) (8*((uint32_t)bs->tail - (uint32_t)bs->start) + bs->pos));
82 }
/* 第 82 行到第 85 行以 bit 数为单位返回码流中的 bit 数。
// */
```

```
84 static __inline uint32_t BitstreamLength(Bitstream * const bs)
85 {
86     uint32_t len = (uint32_t)((uint32_t)bs->tail - (uint32_t)bs->start);

88     if (bs->pos)
89     {
90         uint32_t b = bs->buf;
91         BSWAP(b);
92         *bs->tail = b;
```

```
94         len += (bs->pos + 7) / 8;
95     }
```

```
97     return len;
98 }
```

```
/*
```

第 87 行到第 101 行以字节数返回码流长度。

第 91 行到第 98 行在码流 bit 数不为零时，直接补零到一个完整字节修正长度值。

```
// */
```

```
100 static const int stuffing_codes[8] =
101 {
102     // nbits    stuffing code
103     0,          // 1        0
104     1,          // 2        01
105     3,          // 3        011
106     7,          // 4        0111
107     0xf,        // 5        01111
108     0x1f,       // 6        011111
109     0x3f,       // 7        0111111
110     0x7f,       // 8        01111111
111 };
```

```
113 static __inline void BitstreamPad(Bitstream * const bs)
114 {
115     int bits = 8 - (bs->pos % 8);
116     if (bits < 8)
117         BitstreamPutBits(bs, stuffing_codes[bits - 1], bits);
118 }
```

```
120 static __inline void BitstreamPadAlways(Bitstream * const bs)
121 {
122     int bits = 8 - (bs->pos % 8);
123     BitstreamPutBits(bs, stuffing_codes[bits - 1], bits);
124 }
```

/* 第 116 行到第 127 行，在码流中补 bit 串到完整字节，需要补的 bit 串值由数组 stuffing_codes 确定，没有补零是避免在码流中误生成关键字(0x000001)。

```
// */
```

```
126 #endif
```


8.8 bitstream.c 文件

8.8.1 功能描述

生成 VOL 头和 VOP 头的两个函数及一个辅助函数。按照 MPEG4 标准写有关的位生成相关的码流即可，对照标准很容易理解。

8.8.2 文件注释

```
1  #include "../portab.h"
2  #include "../global.h"
3  #include "../encoder.h"

5  #include "bitstream.h"

7  static uint32_t __inline log2bin(uint32_t value)
8  {
9      int n = 0;

11     while (value)
12     {
13         value >>= 1;
14         n++;
15     }
16     return n;
17 }

/* 第 7 行到第 17 行计算用二进制表示一个数需要的最少 bit 位。
   只需要计算最高位的 bit 位数字即可。
// */

19 void BitstreamWriteVolHeader(Bitstream* const bs, const MBParam* pParam,
20                             const FRAMEINFO* const frame)
21 {
22     BitstreamPutBits(bs, 0x00000100, 32);
23     BitstreamPutBits(bs, 0x00000120, 32);

25     BitstreamPutBit0(bs);          // random_accessible_vol
26     BitstreamPutBits(bs, 1, 8);    // video_object_type_indication

28     BitstreamPutBit0(bs);          // is_object_layer_identified (0=not given)

30     BitstreamPutBits(bs, 1, 4);    // aspect_ratio_info (1=1:1)
```

```
31     BitstreamPutBit1(bs);          // vol_control_parameters
32     BitstreamPutBits(bs, 1, 2); // chroma_format 1="4:2:0"

34     BitstreamPutBit1(bs);  // low_delay

36     BitstreamPutBit0(bs);  // vbv_parameters (0=not given)

38     BitstreamPutBits(bs, 0, 2); // video_object_layer_shape (0=rectangular)

40     WRITE_MARKER();

42     BitstreamPutBits(bs, pParam->fbase, 16);

44     WRITE_MARKER();

46     if (pParam->fincr>0)
47     {
48         BitstreamPutBit1(bs);  // fixed_vop_rate = 1 // fixed_vop_time_increment
49         BitstreamPutBits(bs, pParam->fincr, MAX(log2bin(pParam->fbase-1), 1));
50     }
51     else
52     {
53         BitstreamPutBit0(bs);  // fixed_vop_rate = 0
54     }

56     WRITE_MARKER();
57     BitstreamPutBits(bs, pParam->width, 13);  // width
58     WRITE_MARKER();
59     BitstreamPutBits(bs, pParam->height, 13); // height
60     WRITE_MARKER();

62     BitstreamPutBit0(bs);  // interlace
63     BitstreamPutBit1(bs);  // obmc_disable(overlapped block motion compensation)

65     BitstreamPutBit0(bs);  // sprite_enable==off

67     BitstreamPutBit0(bs);  // not_8_bit

69     BitstreamPutBit0(bs);  // quant_type  0=h.263  1=mpeg4(quantizer tables)
```

```
71     BitstreamPutBit1(bs);        // complexity_estimation_disable
72     BitstreamPutBit1(bs);        // resync_marker_disable
73     BitstreamPutBit0(bs);        // data_partitioned

75     BitstreamPutBit0(bs);        // scalability

77     BitstreamPadAlways(bs);      // next_start_code()
78 }
/* 第 19 行到第 78 行按照 MPEG4 标准写 VOL 头，因为不支持某些特性，就删掉很多的判断项，
   程序看起来相对要简洁一些。 这部分代码对照着 MPEG4 标准一看就懂。
   // */

80 void BitstreamWriteVopHeader(Bitstream* const bs, const MBParam* pParam,
81                               const FRAMEINFO* const frame, unsigned int quant)
82 {
83     BitstreamPutBits(bs, 0x000001b6, 32);

85     BitstreamPutBits(bs, frame->coding_type, 2);

87     BitstreamPutBit1(bs);

89     BitstreamPutBit0(bs);

91     WRITE_MARKER();

93     // time_increment: value=nth_of_sec, nbits = log2(resolution)
94     BitstreamPutBits(bs, 0, MAX(log2bin(pParam->fbase-1), 1));

96     WRITE_MARKER();

98     BitstreamPutBits(bs, 1, 1); // vop_coded

100    if ( frame->coding_type == P_VOP)
101        BitstreamPutBits(bs, frame->rounding_type, 1);

103    BitstreamPutBits(bs, 0, 3); // intra_dc_vlc_threshold

105    BitstreamPutBits(bs, quant, 5); // quantizer

107    if (frame->coding_type != I_VOP)
```

```
108         BitstreamPutBits(bs, frame->fcode, 3); // forward_fixed_code
109     }
```

/* 第 80 行到第 111 行按照 MPEG4 标准写 VOP 头，因为不支持某些特性，就删掉很多的判断项，程序看起来相对要简洁一些。

除第 96 行代码外，这部分代码对照着 MPEG4 标准一看就懂。

我们一直认为时钟是一个非常重要的参量，不能出错。在非实时情况下，

我们保存成 AVI 文件，由 AVI 文件来保证时钟正确。在实时情况下，

解码端收到一帧解码一帧显示一帧，时钟误差在可接受范围内。因此，MPEG4 视频码流

里面和时钟有关的字段不太重要，我们就简化处理成 0，所以第 96 行和标准有一点点不同。

```
// */
```

附 录

附录 A 程序用到的部分 MMX/SSE2 汇编指令

MOVLHPS:Move Packed Single-Precision Floating-Point Values Low to High

SRC	SRC (127-64)	SRC (63-0)
DST	DST (127-64)	DST (63-0)
DST	SRC (63-0)	DST (63-0)

MOVLPS:Move Low Packed Single-Precision Floating-Point Values

SRC	SRC (127-64)	SRC (63-0)
DST	DST (127-64)	DST (63-0)
DST	DST (127-64)	SRC (63-0)

PADDB / PADDW / PADDD:Add Packed Integers(PADDB / W / D)

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
DST	Wrap (X7+Y7)	Wrap (X6+Y6)	Wrap (X5+Y5)	Wrap (X4+Y4)	Wrap (X3+Y3)	Wrap (X2+Y2)	Wrap (X1+Y1)	Wrap (X0+Y0)

PADDUSW: Add Packed Unsigned Integers with Unsigned Saturation

SRC	unsigned X7	unsigned X6	unsigned X5	unsigned X4	unsigned X3	unsigned X2	unsigned X1	unsigned X0
DST	unsigned Y7	unsigned Y6	unsigned Y5	unsigned Y4	unsigned Y3	unsigned Y2	unsigned Y1	unsigned Y0
DST	Sat (X7+Y7)	Sat (X6+Y6)	Sat (X5+Y5)	Sat (X4+Y4)	Sat (X3+Y3)	Sat (X2+Y2)	Sat (X1+Y1)	Sat (X0+Y0)

PSUBB / PSUBW / PSUBD:Subtract Packed Integers(PSUBB / W / D)

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
DST	Wrap (X7-Y7)	Wrap (X6-Y6)	Wrap (X5-Y5)	Wrap (X4-Y4)	Wrap (X3-Y3)	Wrap (X2-Y2)	Wrap (X1-Y1)	Wrap (X0-Y0)

PSUBUSB / W: Subtract Packed Unsigned Integers with Unsigned Saturation

SRC	unsigned X7	unsigned X6	unsigned X5	unsigned X4	unsigned X3	unsigned X2	unsigned X1	unsigned X0
DST	unsigned Y7	unsigned Y6	unsigned Y5	unsigned Y4	unsigned Y3	unsigned Y2	unsigned Y1	unsigned Y0
DST	Sat (X7-Y7)	Sat (X6-Y6)	Sat (X5-Y5)	Sat (X4-Y4)	Sat (X3-Y3)	Sat (X2-Y2)	Sat (X1-Y1)	Sat (X0-Y0)

PXOR: Logical Exclusive OR

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
DST	X7 XOR Y7	X6 XOR Y6	X5 XOR Y5	X4 XOR Y4	X3 XOR Y3	X2 XOR Y2	X1 XOR Y1	X0 XOR Y0

PSRLW / PSRLD / PSRLQ: Shift Packed Data Right Logical (with Zero Extension)

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	X7>>count	X6>>count	X5>>count	X4>>count	X3>>count	X2>>count	X1>>count	X0>>count

PANDN: Logical AND NOT

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
DST	(NOT Y7) & X7	(NOT Y6) & X6	(NOT Y5) & X5	(NOT Y4) & X4	(NOT Y3) & X3	(NOT Y2) & X2	(NOT Y1) & X1	(NOT Y0) & X0

PMULHW: Multiply Packed Signed Integers and Store High Result

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
TMP	X7*Y7,[31,0]	X6*Y6,[31,0]	X5*Y5,[31,0]	X4*Y4,[31,0]	X3*Y3,[31,0]	X2*Y2,[31,0]	X1*Y1,[31,0]	X0*Y0,[31,0]
DST	TMP7[31, 16]	TMP6[31, 16]	TMP5[31, 16]	TMP4[31, 16]	TMP3[31, 16]	TMP2[31, 16]	TMP1[31, 16]	TMP0[31, 16]

PMAWSW:Maximum of Packed Signed Word Integers

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
DST	if (X7>Y7) X7 else Y7	if (X6>Y6) X6 else Y6	if (X5>Y5) X5 else Y5	if (X4>Y4) X4 else Y4	if (X3>Y3) X3 else Y3	if (X2>Y2) X2 else Y2	if (X1>Y1) X1 else Y1	if (X0>Y0) X0 else Y0

PMINSW:Minimum of Packed Signed Word Integers

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
DST	if (X7<Y7) X7 else Y7	if (X6<Y6) X6 else Y6	if (X5<Y5) X5 else Y5	if (X4<Y4) X4 else Y4	if (X3<Y3) X3 else Y3	if (X2<Y2) X2 else Y2	if (X1<Y1) X1 else Y1	if (X0<Y0) X0 else Y0

PUNPCKHBW / PUNPCKHWD / PUNPCKHDQ:Unpack High Data

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
DST	Y7	X7	Y6	X6	Y5	X5	Y4	X4

PUNPCKLBW / PUNPCKLWD / PUNPCKLDQ:Unpack Low Data

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
DST	Y3	X3	Y2	X2	Y1	X1	Y0	X0

PACKUSWB:Pack with Unsigned Saturation

SRC	X3	X2	X1	X0
DST	Y3	Y2	Y1	Y1

DST	unsign Sat(Y3)	unsign Sat(Y2)	unsign Sat(Y1)	unsign Sat(Y0)	unsign Sat(X3)	unsign Sat(X2)	unsign Sat(X1)	unsign Sat(X0)
-----	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

PACKSSWB: Pack with Signed Saturation

SRC	X3	X2	X1	X0
DST	Y3	Y2	Y1	Y1

DST	signSat(Y3)	signSat(Y2)	signSat(Y1)	signSat(Y0)	signSat(X3)	signSat(X2)	signSat(X1)	signSat(X0)
-----	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

PAVGB / PAVGW: Average Packed Integers

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

DST	$(Y7+X7)/2$	$(Y6+X6)/2$	$(Y5+X5)/2$	$(Y4+X4)/2$	$(Y3+X3)/2$	$(Y2+X2)/2$	$(Y1+X1)/2$	$(Y0+X0)/2$
-----	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

PSADBW: Compute Sum of Absolute Differences

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
TMP	ABS(X7-Y7)	ABS(X6-Y6)	ABS(X5-Y5)	ABS(X4-Y4)	ABS(X3-Y3)	ABS(X2-Y2)	ABS(X1-Y1)	ABS(X0-Y0)

DST							SUM(TMP7...TMP0)	
-----	--	--	--	--	--	--	------------------	--

PCMPGTW: Compare Packed Signed Integers for Greater Than

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

DST	if(X7>Y7) 0 else 0xFFFF	if(X6>Y6) 0 else 0xFFFF	if(X5>Y5) 0 else 0xFFFF	if(X4>Y4) 0 else 0xFFFF	if(X3>Y3) 0 else 0xFFFF	if(X2>Y2) 0 else 0xFFFF	if(X1>Y1) 0 else 0xFFFF	if(X0>Y0) 0 else 0xFFFF
-----	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------

PMADDWD: Multiply and Add Packed Integers

SRC	X7	X6	X5	X4	X3	X2	X1	X0
DST	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

DST	Y7*X7+Y6*X6		Y5*X5+Y4*X4		Y3*X3+Y2*X2		Y1*X1+Y0*X0	
-----	-------------	--	-------------	--	-------------	--	-------------	--

PEXTRW:Extract Word

Instruction	Description
PEXTRW <i>r32</i> , <i>mm</i> , <i>imm8</i>	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>r32</i> .
PEXTRW <i>r32</i> , <i>xmm</i> , <i>imm8</i>	Extract the word specified by <i>imm8</i> from <i>xmm</i> and move it to a <i>r32</i> .

PSHUFLW:Shuffle Packed Low Words

DEST[15-0] ← (SRC (ORDER[1-0] * 16)) [15-0]

DEST[31-16] ← (SRC (ORDER[3-2] * 16)) [15-0]

DEST[47-32] ← (SRC (ORDER[5-4] * 16)) [15-0]

DEST[63-48] ← (SRC (ORDER[7-6] * 16)) [15-0]

DEST[127-64] ← (SRC[127-64]

PSHUFHW:Shuffle Packed High Words

DEST[63-0] ← (SRC[63-0]

DEST[79-64] ← (SRC (ORDER[1-0] * 16)) [79-64]

DEST[95-80] ← (SRC (ORDER[3-2] * 16)) [79-64]

DEST[111-96] ← (SRC (ORDER[5-4] * 16)) [79-64]

DEST[127-112] ← (SRC (ORDER[7-6] * 16)) [79-64]

附录 B 备忘录

20080416: Create by yangshuliang tinck

20080507: Change some mistake press