原 FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-TU

2015年06月16日 17:14:42

===

HEVC源代码分析文章列表：

【解码 -libavcodec HEVC 解码器】

===

本文分析FFmpeg的libavcodec中的HEVC解码器的CTU解码（CTU Decode）部分的源代码。FFmpeg的HEVC解码器调用hls_decode_entry()函数完成了Slice解码工作。hls_decode_entry()则调用了hls_coding_quadtree()完成了CTU解码工作。由于CTU解码部分的内容比较多，因此将这一部分内容拆分成两篇文章：一篇文章记录PU的解码，另一篇文章记录TU解码。本文记录TU的解码过程。

函数调用关系图

FFmpeg HEVC解码器的CTU解码（CTU Decoder）部分在整个HEVC解码器中的位置如下图所示。

单击查看更清晰的大图

CTU解码（CTU Decoder）部分的函数调用关系如下图所示。

单击查看更清晰的大图

从图中可以看出，CTU解码模块对应的函数是hls_coding_quadtree()。该函数是一个递归调用的函数，可以按照四叉树的句法格式解析CTU并获得其中的CU。对于每个CU会调用hls_coding_unit()进行解码。

hls_coding_unit()会调用hls_prediction_unit()对CU中的PU进行处理。hls_prediction_unit()调用luma_mc_uni()对亮度单向预测块进行运动补偿处理，调用chroma_mc_uni()对色度单向预测块进行运动补偿处理，调用luma_mc_bi()对亮度单向预测块进行运动补偿处理。

hls_coding_unit()会调用hls_transform_tree()对CU中的TU进行处理。hls_transform_tree()是一个递归调用的函数，可以按照四叉树的句法格式解析并获得其中的TU。对于每一个TU会调用hls_transform_unit()进行解码。hls_transform_unit()会进行帧内预测，并且调用ff_hevc_hls_residual_coding()解码DCT残差数据。

hls_decode_entry()

hls_decode_entry()是FFmpeg HEVC解码器中Slice解码的入口函数。该函数的定义如下所示。

[cpp]

```cpp
//解码入口函数
static int hls_decode_entry(AVCodecContext *avctxt, void *isFilterThread)
{
    HEVCContext *s  = avctxt->priv_data;
    //CTB尺寸
    int ctb_size    = 1 << s->sps->log2_ctb_size;
    int more_data   = 1;
    int x_ctb       = 0;
    int y_ctb       = 0;
    int ctb_addr_ts = s->pps->ctb_addr_rs_to_ts[s->sh.slice_ctb_addr_rs];

    if (!ctb_addr_ts && s->sh.dependent_slice_segment_flag) {
        av_log(s->avctx, AV_LOG_ERROR, "Impossible initial tile.\n");
        return AVERROR_INVALIDDATA;
```

```
    }

    if (s->sh.dependent_slice_segment_flag) {
        int prev_rs = s->pps->ctb_addr_ts_to_rs[ctb_addr_ts - 1];
        if (s->tab_slice_address[prev_rs] != s->sh.slice_addr) {
            av_log(s->avctx, AV_LOG_ERROR, "Previous slice segment missing\n");
            return AVERROR_INVALIDDATA;
        }
    }

    while (more_data && ctb_addr_ts < s->sps->ctb_size) {
        int ctb_addr_rs = s->pps->ctb_addr_ts_to_rs[ctb_addr_ts];
        //CTB的位置x和y
        x_ctb = (ctb_addr_rs % ((s->sps->width + ctb_size - 1) >> s->sps->log2_ctb_size)) << s->sps->log2_ctb_size;
        y_ctb = (ctb_addr_rs / ((s->sps->width + ctb_size - 1) >> s->sps->log2_ctb_size)) << s->sps->log2_ctb_size;
        //初始化周围的参数
        hls_decode_neighbour(s, x_ctb, y_ctb, ctb_addr_ts);
        //初始化CABAC
        ff_hevc_cabac_init(s, ctb_addr_ts);
        //样点自适应补偿参数
        hls_sao_param(s, x_ctb >> s->sps->log2_ctb_size, y_ctb >> s->sps->log2_ctb_size);

        s->deblock[ctb_addr_rs].beta_offset = s->sh.beta_offset;
        s->deblock[ctb_addr_rs].tc_offset   = s->sh.tc_offset;
        s->filter_slice_edges[ctb_addr_rs]  = s->sh.slice_loop_filter_across_slices_enabled_flag;
        /*
         * CU示意图
         *
         * 64x64块
         *
         * 深度d=0
         * split_flag=1时候划分为4个32x32
         *
         * +--------+--------+--------+--------+--------+--------+--------+--------+
         * |                                                                       |
         * |                                   |                                   |
         * |                                                                       |
         * +                                   |                                   +
         * |                                                                       |
         * |                                   |                                   |
         * |                                                                       |
         * +                                   |                                   +
         * |                                                                       |
         * |                                   |                                   |
         * |                                                                       |
         * +                                   |                                   +
         * |                                                                       |
         * |                                   |                                   |
         * |                                                                       |
         * + --  --  --  --  --  --  --  --  --+ --  --  --  --  --  --  --  --  --+
         * |                                   |                                   |
         * |                                                                       |
         * |                                   |                                   |
         * +                                                                       +
         * |                                   |                                   |
         * |                                                                       |
         * |                                   |                                   |
         * +                                                                       +
         * |                                   |                                   |
         * |                                                                       |
         * |                                   |                                   |
         * +                                                                       +
         * |                                   |                                   |
         * |                                                                       |
         * |                                   |                                   |
         * +--------+--------+--------+--------+--------+--------+--------+--------+
         *
         *
         * 32x32 块
         * 深度d=1
         * split_flag=1时候划分为4个16x16
         *
         * +--------+--------+--------+--------+
         * |                                   |
         * |                 |                 |
         * |                                   |
         * +                 |                 +
         * |                                   |
         * |                 |                 |
         * |                                   |
         * + --  --  --  --  + --  --  --  --  +
         * |                                   |
         * |                 |                 |
         * |                                   |
         * +                 |                 +
         * |                                   |
         * |                 |                 |
         * |                                   |
         * +--------+--------+--------+--------+
         *
         *
```

```cpp
         * 16x16 块
         * 深度d=2
         * split_flag=1时候划分为4个8x8
         *
         * +--------+--------+
         * |                 |
         * |        |        |
         * |                 |
         * +  --   --+ --   -- +
         * |                 |
         * |        |        |
         * |                 |
         * +--------+--------+
         *
         *
         * 8x8块
         * 深度d=3
         * split_flag=1时候划分为4个4x4
         *
         * +----+----+
         * |    |    |
         * + -- + -- +
         * |    |    |
         * +----+----+
         *
         */
        /*
         * 解析四叉树结构，并且解码
         *
         * hls_coding_quadtree(HEVCContext *s, int x0, int y0, int log2_cb_size, int cb_depth)中：
         * s：HEVCContext上下文结构体
         * x_ctb：CB位置的x坐标
         * y_ctb：CB位置的y坐标
         * log2_cb_size：CB大小取log2之后的值
         * cb_depth：深度
         *
         */
        more_data = hls_coding_quadtree(s, x_ctb, y_ctb, s->sps->log2_ctb_size, 0);
        if (more_data < 0) {
            s->tab_slice_address[ctb_addr_rs] = -1;
            return more_data;
        }


        ctb_addr_ts++;
        //保存解码信息以供下次使用
        ff_hevc_save_states(s, ctb_addr_ts);
        //去块效应滤波
        ff_hevc_hls_filters(s, x_ctb, y_ctb, ctb_size);
    }

    if (x_ctb + ctb_size >= s->sps->width &&
        y_ctb + ctb_size >= s->sps->height)
        ff_hevc_hls_filter(s, x_ctb, y_ctb, ctb_size);

    return ctb_addr_ts;
}
```

从源代码可以看出，hls_decode_entry()主要调用了2个函数进行解码工作：

（1）调用hls_coding_quadtree()解码CTU。其中包含了PU和TU的解码。

（2）调用ff_hevc_hls_filters()进行滤波。其中包含了去块效应滤波和SAO滤波。

本文分析第一步的CTU解码过程。

hls_coding_quadtree()

hls_coding_quadtree()用于解析CTU的四叉树句法结构。该函数的定义如下所示。

[cpp]

```cpp
/*
 * 解析四叉树结构，并且解码
 * 注意该函数是递归调用
 * 注释和处理：雷霄骅
 *
 *
 * s：HEVCContext上下文结构体
 * x_ctb：CB位置的x坐标
 * y_ctb：CB位置的y坐标
 * log2_cb_size：CB大小取log2之后的值
 * cb_depth：深度
 *
 */
static int hls_coding_quadtree(HEVCContext *s, int x0, int y0,
                               int log2_cb_size, int cb_depth)
{
    HEVCLocalContext *lc = s->HEVClc;
    //CB的大小,split flag=0
    //log2_cb_size为CB大小取log之后的结果
```

```
    //log2_cb_size为CB大小取log之后的结果
    const int cb_size    = 1 << log2_cb_size;
    int ret;
    int qp_block_mask = (1<<(s->sps->log2_ctb_size - s->pps->diff_cu_qp_delta_depth)) - 1;
    int split_cu;

    lc->ct_depth = cb_depth;
    if (x0 + cb_size <= s->sps->width  &&
        y0 + cb_size <= s->sps->height &&
        log2_cb_size > s->sps->log2_min_cb_size) {
        split_cu = ff_hevc_split_coding_unit_flag_decode(s, cb_depth, x0, y0);
    } else {
        split_cu = (log2_cb_size > s->sps->log2_min_cb_size);
    }
    if (s->pps->cu_qp_delta_enabled_flag &&
        log2_cb_size >= s->sps->log2_ctb_size - s->pps->diff_cu_qp_delta_depth) {
        lc->tu.is_cu_qp_delta_coded = 0;
        lc->tu.cu_qp_delta          = 0;
    }

    if (s->sh.cu_chroma_qp_offset_enabled_flag &&
        log2_cb_size >= s->sps->log2_ctb_size - s->pps->diff_cu_chroma_qp_offset_depth) {
        lc->tu.is_cu_chroma_qp_offset_coded = 0;
    }

    if (split_cu) {
        //如果CU还可以继续划分，则继续解析划分后的CU
        //注意这里是递归调用


        //CB的大小,split flag=1
        const int cb_size_split = cb_size >> 1;

        /*
         * (x0, y0)  (x1, y0)
         *     +--------+--------+
         *     |                 |
         *     |        |        |
         *     |                 |
         *     +  --  --+ --  -- +
         * (x0, y1)  (x1, y1)    |
         *     |        |        |
         *     |                 |
         *     +--------+--------+
         *
         */
        const int x1 = x0 + cb_size_split;
        const int y1 = y0 + cb_size_split;

        int more_data = 0;

        //注意：
        //CU大小减半，log2_cb_size-1
        //深度d加1，cb_depth+1
        more_data = hls_coding_quadtree(s, x0, y0, log2_cb_size - 1, cb_depth + 1);
        if (more_data < 0)
            return more_data;

        if (more_data && x1 < s->sps->width) {
            more_data = hls_coding_quadtree(s, x1, y0, log2_cb_size - 1, cb_depth + 1);
            if (more_data < 0)
                return more_data;
        }
        if (more_data && y1 < s->sps->height) {
            more_data = hls_coding_quadtree(s, x0, y1, log2_cb_size - 1, cb_depth + 1);
            if (more_data < 0)
                return more_data;
        }
        if (more_data && x1 < s->sps->width &&
            y1 < s->sps->height) {
            more_data = hls_coding_quadtree(s, x1, y1, log2_cb_size - 1, cb_depth + 1);
            if (more_data < 0)
                return more_data;
        }

        if(((x0 + (1<<log2_cb_size)) & qp_block_mask) == 0 &&
            ((y0 + (1<<log2_cb_size)) & qp_block_mask) == 0)
            lc->qPy_pred = lc->qp_y;

        if (more_data)
            return ((x1 + cb_size_split) < s->sps->width ||
                    (y1 + cb_size_split) < s->sps->height);
        else
            return 0;
    } else {

        /*
         * (x0, y0)
         *     +--------+--------+
         *     |                 |
         *     |                 |
         *     |                 |
```

```
         *     +                 +
         *     |                 |
         *     |                 |
         *     |                 |
         *     +--------+--------+
         *
         */
        //注意处理的是不可划分的CU单元
        //处理CU单元-真正的解码
        ret = hls_coding_unit(s, x0, y0, log2_cb_size);
        if (ret < 0)
            return ret;
        if ((!((x0 + cb_size) %
               (1 << (s->sps->log2_ctb_size))) ||
             (x0 + cb_size >= s->sps->width)) &&
            (!((y0 + cb_size) %
               (1 << (s->sps->log2_ctb_size))) ||
             (y0 + cb_size >= s->sps->height))) {
            int end_of_slice_flag = ff_hevc_end_of_slice_flag_decode(s);
            return !end_of_slice_flag;
        } else {
            return 1;
        }
    }

    return 0;
}
```

从源代码可以看出，hls_coding_quadtree()首先调用ff_hevc_split_coding_unit_flag_decode()判断当前CU是否还需要划分。如果需要划分的话，就会递归调用4次hls_coding_quadtree()分别对4个子块继续进行四叉树解析；如果不需要划分，就会调用hls_coding_unit()对CU进行解码。总而言之，hls_coding_quadtree()会解析出来一个CTU中的所有CU，并且对每一个CU逐一调用hls_coding_unit()进行解码。一个CTU中CU的解码顺序如下图所示。图中a, b, c …即代表了的先后顺序。

hls_coding_unit()

hls_coding_unit()用于解码一个CU。该函数的定义如下所示。

[cpp]

```cpp
//处理CU单元-真正的解码
//注释和处理：雷霄骅
static int hls_coding_unit(HEVCContext *s, int x0, int y0, int log2_cb_size)
{
    //CB大小
    int cb_size          = 1 << log2_cb_size;
    HEVCLocalContext *lc = s->HEVClc;
    int log2_min_cb_size = s->sps->log2_min_cb_size;
    int length           = cb_size >> log2_min_cb_size;
    int min_cb_width     = s->sps->min_cb_width;
    //以最小的CB为单位（例如4x4）的时候，当前CB的位置——x坐标和y坐标
    int x_cb             = x0 >> log2_min_cb_size;
    int y_cb             = y0 >> log2_min_cb_size;
    int idx              = log2_cb_size - 2;
    int qp_block_mask    = (1<<(s->sps->log2_ctb_size - s->pps->diff_cu_qp_delta_depth)) - 1;
    int x, y, ret;

    //设置CU的属性值
    lc->cu.x                = x0;
    lc->cu.y                = y0;
    lc->cu.pred_mode        = MODE_INTRA;
    lc->cu.part_mode        = PART_2Nx2N;
    lc->cu.intra_split_flag = 0;

    SAMPLE_CTB(s->skip_flag, x_cb, y_cb) = 0;

    for (x = 0; x < 4; x++)
        lc->pu.intra_pred_mode[x] = 1;
    if (s->pps->transquant_bypass_enable_flag) {
        lc->cu.cu_transquant_bypass_flag = ff_hevc_cu_transquant_bypass_flag_decode(s);
        if (lc->cu.cu_transquant_bypass_flag)
            set_deblocking_bypass(s, x0, y0, log2_cb_size);
    } else
        lc->cu.cu_transquant_bypass_flag = 0;

    if (s->sh.slice_type != I_SLICE) {
        //Skip类型
        uint8_t skip_flag = ff_hevc_skip_flag_decode(s, x0, y0, x_cb, y_cb);
        //设置到skip_flag缓存中
        x = y_cb * min_cb_width + x_cb;
        for (y = 0; y < length; y++) {
            memset(&s->skip_flag[x], skip_flag, length);
            x += min_cb_width;
        }
```

```
        lc->cu.pred_mode = skip_flag ? MODE_SKIP : MODE_INTER;
    } else {
        x = y_cb * min_cb_width + x_cb;
        for (y = 0; y < length; y++) {
            memset(&s->skip_flag[x], 0, length);
            x += min_cb_width;
        }
    }

    if (SAMPLE_CTB(s->skip_flag, x_cb, y_cb)) {
        hls_prediction_unit(s, x0, y0, cb_size, cb_size, log2_cb_size, 0, idx);
        intra_prediction_unit_default_value(s, x0, y0, log2_cb_size);

        if (!s->sh.disable_deblocking_filter_flag)
            ff_hevc_deblocking_boundary_strengths(s, x0, y0, log2_cb_size);
    } else {
        int pcm_flag = 0;

        //读取预测模式（非 I Slice）
        if (s->sh.slice_type != I_SLICE)
            lc->cu.pred_mode = ff_hevc_pred_mode_decode(s);

        //不是帧内预测模式的时候
        //或者已经是最小CB的时候
        if (lc->cu.pred_mode != MODE_INTRA ||
            log2_cb_size == s->sps->log2_min_cb_size) {
            //读取CU分割模式
            lc->cu.part_mode        = ff_hevc_part_mode_decode(s, log2_cb_size);
            lc->cu.intra_split_flag = lc->cu.part_mode == PART_NxN &&
                                      lc->cu.pred_mode == MODE_INTRA;
        }

        if (lc->cu.pred_mode == MODE_INTRA) {
            //帧内预测模式

            //PCM方式编码，不常见
            if (lc->cu.part_mode == PART_2Nx2N && s->sps->pcm_enabled_flag &&
                log2_cb_size >= s->sps->pcm.log2_min_pcm_cb_size &&
                log2_cb_size <= s->sps->pcm.log2_max_pcm_cb_size) {
                pcm_flag = ff_hevc_pcm_flag_decode(s);
            }
            if (pcm_flag) {
                intra_prediction_unit_default_value(s, x0, y0, log2_cb_size);
                ret = hls_pcm_sample(s, x0, y0, log2_cb_size);
                if (s->sps->pcm.loop_filter_disable_flag)
                    set_deblocking_bypass(s, x0, y0, log2_cb_size);

                if (ret < 0)
                    return ret;
            } else {
                //帧内预测
                intra_prediction_unit(s, x0, y0, log2_cb_size);
            }
        } else {
            //帧间预测模式
            intra_prediction_unit_default_value(s, x0, y0, log2_cb_size);

            //帧间模式一共有8种划分模式

            switch (lc->cu.part_mode) {
            case PART_2Nx2N:
                /*
                 * PART_2Nx2N:
                 * +--------+--------+
                 * |                 |
                 * |                 |
                 * |                 |
                 * +        +        +
                 * |                 |
                 * |                 |
                 * |                 |
                 * +--------+--------+
                 */
                //处理PU单元-运动补偿
                hls_prediction_unit(s, x0, y0, cb_size, cb_size, log2_cb_size, 0, idx);
                break;
            case PART_2NxN:
                /*
                 * PART_2NxN:
                 * +--------+--------+
                 * |                 |
                 * |                 |
                 * |                 |
                 * +--------+--------+
                 * |                 |
                 * |                 |
                 * |                 |
                 * +--------+--------+
                 *
                 */
                /*
                 * hls_prediction_unit()参数：
```

```
             * hls_prediction_unit()参数：
             * x0 : PU左上角x坐标
             * y0 : PU左上角y坐标
             * nPbW : PU宽度
             * nPbH : PU高度
             * log2_cb_size : CB大小取log2()的值
             * partIdx : PU的索引号-分成4个块的时候取0-3，分成两个块的时候取0和1
             */
            //上
            hls_prediction_unit(s, x0, y0,               cb_size, cb_size / 2, log2_cb_size, 0, idx);
            //下
            hls_prediction_unit(s, x0, y0 + cb_size / 2, cb_size, cb_size / 2, log2_cb_size, 1, idx);
            break;
        case PART_Nx2N:
            /*
             * PART_Nx2N:
             * +--------+--------+
             * |        |        |
             * |        |        |
             * |        |        |
             * +        +        +
             * |        |        |
             * |        |        |
             * |        |        |
             * +--------+--------+
             *
             */
            //左
            hls_prediction_unit(s, x0,               y0, cb_size / 2, cb_size, log2_cb_size, 0, idx - 1);
            //右
            hls_prediction_unit(s, x0 + cb_size / 2, y0, cb_size / 2, cb_size, log2_cb_size, 1, idx - 1);
            break;
        case PART_2NxnU:
            /*
             * PART_2NxnU (Upper) :
             * +--------+--------+
             * |                 |
             * +--------+--------+
             * |                 |
             * +        +        +
             * |                 |
             * |                 |
             * |                 |
             * +--------+--------+
             *
             */
            //上
            hls_prediction_unit(s, x0, y0,               cb_size, cb_size     / 4, log2_cb_size, 0, idx);
            //下
            hls_prediction_unit(s, x0, y0 + cb_size / 4, cb_size, cb_size * 3 / 4, log2_cb_size, 1, idx);
            break;
        case PART_2NxnD:
            /*
             * PART_2NxnD (Down) :
             * +--------+--------+
             * |                 |
             * |                 |
             * |                 |
             * +        +        +
             * |                 |
             * +--------+--------+
             * |                 |
             * +--------+--------+
             *
             */
            //上
            hls_prediction_unit(s, x0, y0,                   cb_size, cb_size * 3 / 4, log2_cb_size, 0, idx);
            //下
            hls_prediction_unit(s, x0, y0 + cb_size * 3 / 4, cb_size, cb_size     / 4, log2_cb_size, 1, idx);
            break;
        case PART_nLx2N:
            /*
             * PART_nLx2N (Left):
             * +----+---+--------+
             * |    |            |
             * |    |            |
             * |    |            |
             * +    +   +        +
             * |    |            |
             * |    |            |
             * |    |            |
             * +----+---+--------+
             *
             */
            //左
            hls_prediction_unit(s, x0,               y0, cb_size     / 4, cb_size, log2_cb_size, 0, idx - 2);
            //右
            hls_prediction_unit(s, x0 + cb_size / 4, y0, cb_size * 3 / 4, cb_size, log2_cb_size, 1, idx - 2);
            break;
        case PART_nRx2N:
            /*
             * PART_nRx2N (Right):
```

```
                 * +--------+---+----+
                 * |            |    |
                 * |            |    |
                 * |            |    |
                 * +        +   +    +
                 * |            |    |
                 * |            |    |
                 * |            |    |
                 * +--------+---+----+
                 *
                 */
                //左
                hls_prediction_unit(s, x0,                   y0, cb_size * 3 / 4, cb_size, log2_cb_size, 0, idx - 2);
                //右
                hls_prediction_unit(s, x0 + cb_size * 3 / 4, y0, cb_size     / 4, cb_size, log2_cb_size, 1, idx - 2);
                break;
            case PART_NxN:
                /*
                 * PART_NxN:
                 * +--------+--------+
                 * |        |        |
                 * |        |        |
                 * |        |        |
                 * +--------+--------+
                 * |        |        |
                 * |        |        |
                 * |        |        |
                 * +--------+--------+
                 *
                 */
                hls_prediction_unit(s, x0,               y0,               cb_size / 2, cb_size / 2, log2_cb_size, 0, idx - 1);
                hls_prediction_unit(s, x0 + cb_size / 2, y0,               cb_size / 2, cb_size / 2, log2_cb_size, 1, idx - 1);
                hls_prediction_unit(s, x0,               y0 + cb_size / 2, cb_size / 2, cb_size / 2, log2_cb_size, 2, idx - 1);
                hls_prediction_unit(s, x0 + cb_size / 2, y0 + cb_size / 2, cb_size / 2, cb_size / 2, log2_cb_size, 3, idx - 1);
                break;
            }
        }

        if (!pcm_flag) {
            int rqt_root_cbf = 1;

            if (lc->cu.pred_mode != MODE_INTRA &&
                !(lc->cu.part_mode == PART_2Nx2N && lc->pu.merge_flag)) {
                rqt_root_cbf = ff_hevc_no_residual_syntax_flag_decode(s);
            }
            if (rqt_root_cbf) {
                const static int cbf[2] = { 0 };
                lc->cu.max_trafo_depth = lc->cu.pred_mode == MODE_INTRA ?
                                         s->sps->max_transform_hierarchy_depth_intra + lc->cu.intra_split_flag :
                                         s->sps->max_transform_hierarchy_depth_inter;
                //处理TU四叉树
                ret = hls_transform_tree(s, x0, y0, x0, y0, x0, y0,
                                         log2_cb_size,
                                         log2_cb_size, 0, 0, cbf, cbf);
                if (ret < 0)
                    return ret;
            } else {
                if (!s->sh.disable_deblocking_filter_flag)
                    ff_hevc_deblocking_boundary_strengths(s, x0, y0, log2_cb_size);
            }
        }
    }

    if (s->pps->cu_qp_delta_enabled_flag && lc->tu.is_cu_qp_delta_coded == 0)
        ff_hevc_set_qPy(s, x0, y0, log2_cb_size);

    x = y_cb * min_cb_width + x_cb;
    for (y = 0; y < length; y++) {
        memset(&s->qp_y_tab[x], lc->qp_y, length);
        x += min_cb_width;
    }

    if(((x0 + (1<<log2_cb_size)) & qp_block_mask) == 0 &&
       ((y0 + (1<<log2_cb_size)) & qp_block_mask) == 0) {
        lc->qPy_pred = lc->qp_y;
    }

    set_ct_depth(s, x0, y0, log2_cb_size, lc->ct_depth);

    return 0;
}
```

从源代码可以看出，hls_coding_unit()主要进行了两个方面的处理：

（1）调用hls_prediction_unit()处理PU。

（2）调用hls_transform_tree()处理TU树。

本文分析第二个函数hls_transform_tree()中相关的代码。

hls_transform_tree()

hls_transform_tree()用于解析TU四叉树句法。该函数的定义如下所示。

[cpp]

```cpp
//处理TU四叉树
static int hls_transform_tree(HEVCContext *s, int x0, int y0,
                              int xBase, int yBase, int cb_xBase, int cb_yBase,
                              int log2_cb_size, int log2_trafo_size,
                              int trafo_depth, int blk_idx,
                              const int *base_cbf_cb, const int *base_cbf_cr)
{
    HEVCLocalContext *lc = s->HEVClc;
    uint8_t split_transform_flag;
    int cbf_cb[2];
    int cbf_cr[2];
    int ret;

    cbf_cb[0] = base_cbf_cb[0];
    cbf_cb[1] = base_cbf_cb[1];
    cbf_cr[0] = base_cbf_cr[0];
    cbf_cr[1] = base_cbf_cr[1];

    if (lc->cu.intra_split_flag) {
        if (trafo_depth == 1) {
            lc->tu.intra_pred_mode   = lc->pu.intra_pred_mode[blk_idx];
            if (s->sps->chroma_format_idc == 3) {
                lc->tu.intra_pred_mode_c = lc->pu.intra_pred_mode_c[blk_idx];
                lc->tu.chroma_mode_c     = lc->pu.chroma_mode_c[blk_idx];
            } else {
                lc->tu.intra_pred_mode_c = lc->pu.intra_pred_mode_c[0];
                lc->tu.chroma_mode_c     = lc->pu.chroma_mode_c[0];
            }
        }
    } else {
        lc->tu.intra_pred_mode   = lc->pu.intra_pred_mode[0];
        lc->tu.intra_pred_mode_c = lc->pu.intra_pred_mode_c[0];
        lc->tu.chroma_mode_c     = lc->pu.chroma_mode_c[0];
    }

    if (log2_trafo_size <= s->sps->log2_max_trafo_size &&
        log2_trafo_size >  s->sps->log2_min_tb_size    &&
        trafo_depth     < lc->cu.max_trafo_depth       &&
        !(lc->cu.intra_split_flag && trafo_depth == 0)) {
        split_transform_flag = ff_hevc_split_transform_flag_decode(s, log2_trafo_size);
    } else {
        int inter_split = s->sps->max_transform_hierarchy_depth_inter == 0 &&
                          lc->cu.pred_mode == MODE_INTER &&
                          lc->cu.part_mode != PART_2Nx2N &&
                          trafo_depth == 0;
        //split_transform_flag标记当前TU是否要进行四叉树划分
        //为1则需要划分为4个大小相等的，为0则不再划分
        split_transform_flag = log2_trafo_size > s->sps->log2_max_trafo_size ||
                               (lc->cu.intra_split_flag && trafo_depth == 0) ||
                               inter_split;
    }

    if (log2_trafo_size > 2 || s->sps->chroma_format_idc == 3) {
        if (trafo_depth == 0 || cbf_cb[0]) {
            cbf_cb[0] = ff_hevc_cbf_cb_cr_decode(s, trafo_depth);
            if (s->sps->chroma_format_idc == 2 && (!split_transform_flag || log2_trafo_size == 3)) {
                cbf_cb[1] = ff_hevc_cbf_cb_cr_decode(s, trafo_depth);
            }
        }

        if (trafo_depth == 0 || cbf_cr[0]) {
            cbf_cr[0] = ff_hevc_cbf_cb_cr_decode(s, trafo_depth);
            if (s->sps->chroma_format_idc == 2 && (!split_transform_flag || log2_trafo_size == 3)) {
                cbf_cr[1] = ff_hevc_cbf_cb_cr_decode(s, trafo_depth);
            }
        }
    }

    //如果当前TU要进行四叉树划分
    if (split_transform_flag) {
        const int trafo_size_split = 1 << (log2_trafo_size - 1);
        const int x1 = x0 + trafo_size_split;
        const int y1 = y0 + trafo_size_split;

#define SUBDIVIDE(x, y, idx)                                                    \
do {                                                                            \
    ret = hls_transform_tree(s, x, y, x0, y0, cb_xBase, cb_yBase, log2_cb_size, \
                             log2_trafo_size - 1, trafo_depth + 1, idx,         \
                             cbf_cb, cbf_cr);                                   \
    if (ret < 0)                                                                \
        return ret;                                                             \
} while (0)
        //递归调用
        SUBDIVIDE(x0, y0, 0);
        SUBDIVIDE(x1, y0, 1);
```

```
        SUBDIVIDE(x1, y0, 1);
        SUBDIVIDE(x0, y1, 2);
        SUBDIVIDE(x1, y1, 3);

#undef SUBDIVIDE
    } else {
        int min_tu_size      = 1 << s->sps->log2_min_tb_size;
        int log2_min_tu_size = s->sps->log2_min_tb_size;
        int min_tu_width     = s->sps->min_tb_width;
        int cbf_luma         = 1;

        if (lc->cu.pred_mode == MODE_INTRA || trafo_depth != 0 ||
            cbf_cb[0] || cbf_cr[0] ||
            (s->sps->chroma_format_idc == 2 && (cbf_cb[1] || cbf_cr[1]))) {
            cbf_luma = ff_hevc_cbf_luma_decode(s, trafo_depth);
        }
        //处理TU-帧内预测、DCT反变换
        ret = hls_transform_unit(s, x0, y0, xBase, yBase, cb_xBase, cb_yBase,
                                 log2_cb_size, log2_trafo_size,
                                 blk_idx, cbf_luma, cbf_cb, cbf_cr);
        if (ret < 0)
            return ret;
        // TODO: store cbf_luma somewhere else
        if (cbf_luma) {
            int i, j;
            for (i = 0; i < (1 << log2_trafo_size); i += min_tu_size)
                for (j = 0; j < (1 << log2_trafo_size); j += min_tu_size) {
                    int x_tu = (x0 + j) >> log2_min_tu_size;
                    int y_tu = (y0 + i) >> log2_min_tu_size;
                    s->cbf_luma[y_tu * min_tu_width + x_tu] = 1;
                }
        }
        if (!s->sh.disable_deblocking_filter_flag) {
            ff_hevc_deblocking_boundary_strengths(s, x0, y0, log2_trafo_size);
            if (s->pps->transquant_bypass_enable_flag &&
                lc->cu.cu_transquant_bypass_flag)
                set_deblocking_bypass(s, x0, y0, log2_trafo_size);
        }
    }
    return 0;
}
```

从源代码可以看出，hls_transform_tree()首先调用ff_hevc_split_transform_flag_decode()判断当前TU是否还需要划分。如果需要划分的话，就会递归调用4次hls_transform_tree()分别对4个子块继续进行四叉树解析；如果不需要划分，就会调用hls_transform_unit()对TU进行解码。总而言之，hls_transform_tree()会解析出来一个TU树中的所有TU，并且对每一个TU逐一调用hls_transform_unit()进行解码。

hls_transform_unit()

hls_transform_unit()用于解码一个TU，该函数的定义如下所示。

[cpp]

```cpp
//处理TU-帧内预测、DCT反变换
static int hls_transform_unit(HEVCContext *s, int x0, int y0,
                              int xBase, int yBase, int cb_xBase, int cb_yBase,
                              int log2_cb_size, int log2_trafo_size,
                              int blk_idx, int cbf_luma, int *cbf_cb, int *cbf_cr)
{
    HEVCLocalContext *lc = s->HEVClc;
    const int log2_trafo_size_c = log2_trafo_size - s->sps->hshift[1];
    int i;

    if (lc->cu.pred_mode == MODE_INTRA) {
        int trafo_size = 1 << log2_trafo_size;
        ff_hevc_set_neighbour_available(s, x0, y0, trafo_size, trafo_size);

        //注意：帧内预测也是在这里完成
        //帧内预测
        //log2_trafo_size为当前TU大小取log2()之后的值
        s->hpc.intra_pred[log2_trafo_size - 2](s, x0, y0, 0);
    }

    if (cbf_luma || cbf_cb[0] || cbf_cr[0] ||
        (s->sps->chroma_format_idc == 2 && (cbf_cb[1] || cbf_cr[1]))) {
        int scan_idx   = SCAN_DIAG;
        int scan_idx_c = SCAN_DIAG;
        int cbf_chroma = cbf_cb[0] || cbf_cr[0] ||
                         (s->sps->chroma_format_idc == 2 &&
                         (cbf_cb[1] || cbf_cr[1]));

        if (s->pps->cu_qp_delta_enabled_flag && !lc->tu.is_cu_qp_delta_coded) {
            lc->tu.cu_qp_delta = ff_hevc_cu_qp_delta_abs(s);
            if (lc->tu.cu_qp_delta != 0)
                if (ff_hevc_cu_qp_delta_sign_flag(s) == 1)
                    lc->tu.cu_qp_delta = -lc->tu.cu_qp_delta;
            lc->tu.is_cu_qp_delta_coded = 1;

            if (lc->tu.cu_qp_delta < -(26 + s->sps->qp_bd_offset / 2) ||
                lc->tu.cu_qp_delta >  (25 + s->sps->qp_bd_offset / 2)) {
```

```
            lc->tu.cu_qp_delta > (25 + s->sps->qp_bd_offset / 2)) {
                av_log(s->avctx, AV_LOG_ERROR,
                       "The cu_qp_delta %d is outside the valid range "
                       "[%d, %d].\n",
                       lc->tu.cu_qp_delta,
                       -(26 + s->sps->qp_bd_offset / 2),
                        (25 + s->sps->qp_bd_offset / 2));
                return AVERROR_INVALIDDATA;
            }

            ff_hevc_set_qPy(s, cb_xBase, cb_yBase, log2_cb_size);
        }

        if (s->sh.cu_chroma_qp_offset_enabled_flag && cbf_chroma &&
            !lc->cu.cu_transquant_bypass_flag  &&  !lc->tu.is_cu_chroma_qp_offset_coded) {
            int cu_chroma_qp_offset_flag = ff_hevc_cu_chroma_qp_offset_flag(s);
            if (cu_chroma_qp_offset_flag) {
                int cu_chroma_qp_offset_idx  = 0;
                if (s->pps->chroma_qp_offset_list_len_minus1 > 0) {
                    cu_chroma_qp_offset_idx = ff_hevc_cu_chroma_qp_offset_idx(s);
                    av_log(s->avctx, AV_LOG_ERROR,
                        "cu_chroma_qp_offset_idx not yet tested.\n");
                }
                lc->tu.cu_qp_offset_cb = s->pps->cb_qp_offset_list[cu_chroma_qp_offset_idx];
                lc->tu.cu_qp_offset_cr = s->pps->cr_qp_offset_list[cu_chroma_qp_offset_idx];
            } else {
                lc->tu.cu_qp_offset_cb = 0;
                lc->tu.cu_qp_offset_cr = 0;
            }
            lc->tu.is_cu_chroma_qp_offset_coded = 1;
        }

        if (lc->cu.pred_mode == MODE_INTRA && log2_trafo_size < 4) {
            if (lc->tu.intra_pred_mode >= 6 &&
                lc->tu.intra_pred_mode <= 14) {
                scan_idx = SCAN_VERT;
            } else if (lc->tu.intra_pred_mode >= 22 &&
                       lc->tu.intra_pred_mode <= 30) {
                scan_idx = SCAN_HORIZ;
            }

            if (lc->tu.intra_pred_mode_c >=  6 &&
                lc->tu.intra_pred_mode_c <= 14) {
                scan_idx_c = SCAN_VERT;
            } else if (lc->tu.intra_pred_mode_c >= 22 &&
                       lc->tu.intra_pred_mode_c <= 30) {
                scan_idx_c = SCAN_HORIZ;
            }
        }

        lc->tu.cross_pf = 0;

        //读取残差数据，进行反量化，DCT反变换

        //亮度Y
        if (cbf_luma)
            ff_hevc_hls_residual_coding(s, x0, y0, log2_trafo_size, scan_idx, 0);//最后1个参数为颜色分量号

        if (log2_trafo_size > 2 || s->sps->chroma_format_idc == 3) {
            int trafo_size_h = 1 << (log2_trafo_size_c + s->sps->hshift[1]);
            int trafo_size_v = 1 << (log2_trafo_size_c + s->sps->vshift[1]);
            lc->tu.cross_pf  = (s->pps->cross_component_prediction_enabled_flag && cbf_luma &&
                                (lc->cu.pred_mode == MODE_INTER ||
                                 (lc->tu.chroma_mode_c ==  4)));

            if (lc->tu.cross_pf) {
                hls_cross_component_pred(s, 0);
            }
            //色度U
            for (i = 0; i < (s->sps->chroma_format_idc == 2 ? 2 : 1); i++) {
                if (lc->cu.pred_mode == MODE_INTRA) {
                    ff_hevc_set_neighbour_available(s, x0, y0 + (i << log2_trafo_size_c), trafo_size_h, trafo_size_v);
                    s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0 + (i << log2_trafo_size_c), 1);
                }
                if (cbf_cb[i])
                    ff_hevc_hls_residual_coding(s, x0, y0 + (i << log2_trafo_size_c),
                                                log2_trafo_size_c, scan_idx_c, 1);//最后1个参数为颜色分量号
                else
                    if (lc->tu.cross_pf) {
                        ptrdiff_t stride = s->frame->linesize[1];
                        int hshift = s->sps->hshift[1];
                        int vshift = s->sps->vshift[1];
                        int16_t *coeffs_y = (int16_t*)lc->edge_emu_buffer;
                        int16_t *coeffs   = (int16_t*)lc->edge_emu_buffer2;
                        int size = 1 << log2_trafo_size_c;

                        uint8_t *dst = &s->frame->data[1][(y0 >> vshift) * stride +
                                                              ((x0 >> hshift) << s->sps->pixel_shift)];
                        for (i = 0; i < (size * size); i++) {
                            coeffs[i] = ((lc->tu.res_scale_val * coeffs_y[i]) >> 3);
                        }
                        //叠加残差数据
```

```
                        s->hevcdsp.transform_add[log2_trafo_size_c-2](dst, coeffs, stride);
                    }
            }

            if (lc->tu.cross_pf) {
                hls_cross_component_pred(s, 1);
            }
            //色度V
            for (i = 0; i < (s->sps->chroma_format_idc == 2 ? 2 : 1); i++) {
                if (lc->cu.pred_mode == MODE_INTRA) {
                    ff_hevc_set_neighbour_available(s, x0, y0 + (i << log2_trafo_size_c), trafo_size_h, trafo_size_v);
                    s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0 + (i << log2_trafo_size_c), 2);
                }
                //色度Cr
                if (cbf_cr[i])
                    ff_hevc_hls_residual_coding(s, x0, y0 + (i << log2_trafo_size_c),
                                                log2_trafo_size_c, scan_idx_c, 2);
                else
                    if (lc->tu.cross_pf) {
                        ptrdiff_t stride = s->frame->linesize[2];
                        int hshift = s->sps->hshift[2];
                        int vshift = s->sps->vshift[2];
                        int16_t *coeffs_y = (int16_t*)lc->edge_emu_buffer;
                        int16_t *coeffs   = (int16_t*)lc->edge_emu_buffer2;
                        int size = 1 << log2_trafo_size_c;

                        uint8_t *dst = &s->frame->data[2][(y0 >> vshift) * stride +
                                                          ((x0 >> hshift) << s->sps->pixel_shift)];
                        for (i = 0; i < (size * size); i++) {
                            coeffs[i] = ((lc->tu.res_scale_val * coeffs_y[i]) >> 3);
                        }
                        s->hevcdsp.transform_add[log2_trafo_size_c-2](dst, coeffs, stride);
                    }
            }
        } else if (blk_idx == 3) {
            int trafo_size_h = 1 << (log2_trafo_size + 1);
            int trafo_size_v = 1 << (log2_trafo_size + s->sps->vshift[1]);
            for (i = 0; i < (s->sps->chroma_format_idc == 2 ? 2 : 1); i++) {
                if (lc->cu.pred_mode == MODE_INTRA) {
                    ff_hevc_set_neighbour_available(s, xBase, yBase + (i << log2_trafo_size),
                                                    trafo_size_h, trafo_size_v);
                    s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase + (i << log2_trafo_size), 1);
                }
                if (cbf_cb[i])
                    ff_hevc_hls_residual_coding(s, xBase, yBase + (i << log2_trafo_size),
                                                log2_trafo_size, scan_idx_c, 1);
            }
            for (i = 0; i < (s->sps->chroma_format_idc == 2 ? 2 : 1); i++) {
                if (lc->cu.pred_mode == MODE_INTRA) {
                    ff_hevc_set_neighbour_available(s, xBase, yBase + (i << log2_trafo_size),
                                                trafo_size_h, trafo_size_v);
                    s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase + (i << log2_trafo_size), 2);
                }
                if (cbf_cr[i])
                    ff_hevc_hls_residual_coding(s, xBase, yBase + (i << log2_trafo_size),
                                                log2_trafo_size, scan_idx_c, 2);
            }
        }
    } else if (lc->cu.pred_mode == MODE_INTRA) {
        if (log2_trafo_size > 2 || s->sps->chroma_format_idc == 3) {
            int trafo_size_h = 1 << (log2_trafo_size_c + s->sps->hshift[1]);
            int trafo_size_v = 1 << (log2_trafo_size_c + s->sps->vshift[1]);
            ff_hevc_set_neighbour_available(s, x0, y0, trafo_size_h, trafo_size_v);
            s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0, 1);
            s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0, 2);
            if (s->sps->chroma_format_idc == 2) {
                ff_hevc_set_neighbour_available(s, x0, y0 + (1 << log2_trafo_size_c),
                                                trafo_size_h, trafo_size_v);
                s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0 + (1 << log2_trafo_size_c), 1);
                s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0 + (1 << log2_trafo_size_c), 2);
            }
        } else if (blk_idx == 3) {
            int trafo_size_h = 1 << (log2_trafo_size + 1);
            int trafo_size_v = 1 << (log2_trafo_size + s->sps->vshift[1]);
            ff_hevc_set_neighbour_available(s, xBase, yBase,
                                            trafo_size_h, trafo_size_v);
            s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase, 1);
            s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase, 2);
            if (s->sps->chroma_format_idc == 2) {
                ff_hevc_set_neighbour_available(s, xBase, yBase + (1 << (log2_trafo_size)),
                                                trafo_size_h, trafo_size_v);
                s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase + (1 << (log2_trafo_size)), 1);
                s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase + (1 << (log2_trafo_size)), 2);
            }
        }
    }

    return 0;
}
```

从源代码可以看出，如果是帧内CU的话，hls_transform_unit()会调用HEVCPredContext的intra_pred[]()汇编函数进行帧内预测；然后不论帧内预测还是帧间CU都会调用ff_hevc_hls_residual_coding()解码残差数据，并叠加在预测数据上。

ff_hevc_hls_residual_coding()

ff_hevc_hls_residual_coding()用于读取残差数据并进行DCT反变换。该函数的定义如下所示。

[cpp]

```cpp
//读取残差数据，DCT反变换
void ff_hevc_hls_residual_coding(HEVCContext *s, int x0, int y0,
                                int log2_trafo_size, enum ScanType scan_idx,
                                int c_idx)
{
#define GET_COORD(offset, n)                                    \
    do {                                                        \
        x_c = (x_cg << 2) + scan_x_off[n];                      \
        y_c = (y_cg << 2) + scan_y_off[n];                      \
    } while (0)
    HEVCLocalContext *lc = s->HEVClc;
    int transform_skip_flag = 0;

    int last_significant_coeff_x, last_significant_coeff_y;
    int last_scan_pos;
    int n_end;
    int num_coeff = 0;
    int greater1_ctx = 1;

    int num_last_subset;
    int x_cg_last_sig, y_cg_last_sig;

    const uint8_t *scan_x_cg, *scan_y_cg, *scan_x_off, *scan_y_off;

    ptrdiff_t stride = s->frame->linesize[c_idx];
    int hshift = s->sps->hshift[c_idx];
    int vshift = s->sps->vshift[c_idx];
    uint8_t *dst = &s->frame->data[c_idx][(y0 >> vshift) * stride +
                                          ((x0 >> hshift) << s->sps->pixel_shift)];
    int16_t *coeffs = (int16_t*)(c_idx ? lc->edge_emu_buffer2 : lc->edge_emu_buffer);
    uint8_t significant_coeff_group_flag[8][8] = {{0}};
    int explicit_rdpcm_flag = 0;
    int explicit_rdpcm_dir_flag;

    int trafo_size = 1 << log2_trafo_size;
    int i;
    int qp,shift,add,scale,scale_m;
    const uint8_t level_scale[] = { 40, 45, 51, 57, 64, 72 };
    const uint8_t *scale_matrix = NULL;
    uint8_t dc_scale;
    int pred_mode_intra = (c_idx == 0) ? lc->tu.intra_pred_mode :
                                         lc->tu.intra_pred_mode_c;

    memset(coeffs, 0, trafo_size * trafo_size * sizeof(int16_t));

    // Derive QP for dequant
    if (!lc->cu.cu_transquant_bypass_flag) {
        static const int qp_c[] = { 29, 30, 31, 32, 33, 33, 34, 34, 35, 35, 36, 36, 37, 37 };
        static const uint8_t rem6[51 + 4 * 6 + 1] = {
            0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2,
            3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5,
            0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3,
            4, 5, 0, 1, 2, 3, 4, 5, 0, 1
        };

        static const uint8_t div6[51 + 4 * 6 + 1] = {
            0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3,  3,  3,
            3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6,  6,  6,
            7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10,
            10, 10, 11, 11, 11, 11, 11, 11, 12, 12
        };
        int qp_y = lc->qp_y;

        if (s->pps->transform_skip_enabled_flag &&
            log2_trafo_size <= s->pps->log2_max_transform_skip_block_size) {
            transform_skip_flag = ff_hevc_transform_skip_flag_decode(s, c_idx);
        }

        if (c_idx == 0) {
            qp = qp_y + s->sps->qp_bd_offset;
        } else {
            int qp_i, offset;

            if (c_idx == 1)
                offset = s->pps->cb_qp_offset + s->sh.slice_cb_qp_offset +
                         lc->tu.cu_qp_offset_cb;
            else
                offset = s->pps->cr_qp_offset + s->sh.slice_cr_qp_offset +
                         lc->tu.cu_qp_offset_cr;
```

```

            qp_i = av_clip(qp_y + offset, - s->sps->qp_bd_offset, 57);
            if (s->sps->chroma_format_idc == 1) {
                if (qp_i < 30)
                    qp = qp_i;
                else if (qp_i > 43)
                    qp = qp_i - 6;
                else
                    qp = qp_c[qp_i - 30];
            } else {
                if (qp_i > 51)
                    qp = 51;
                else
                    qp = qp_i;
            }

            qp += s->sps->qp_bd_offset;
        }

        shift    = s->sps->bit_depth + log2_trafo_size - 5;
        add      = 1 << (shift-1);
        scale    = level_scale[rem6[qp]] << (div6[qp]);
        scale_m  = 16; // default when no custom scaling lists.
        dc_scale = 16;

        if (s->sps->scaling_list_enable_flag && !(transform_skip_flag && log2_trafo_size > 2)) {
            const ScalingList *sl = s->pps->scaling_list_data_present_flag ?
            &s->pps->scaling_list : &s->sps->scaling_list;
            int matrix_id = lc->cu.pred_mode != MODE_INTRA;

            matrix_id = 3 * matrix_id + c_idx;

            scale_matrix = sl->sl[log2_trafo_size - 2][matrix_id];
            if (log2_trafo_size >= 4)
                dc_scale = sl->sl_dc[log2_trafo_size - 4][matrix_id];
        }
    } else {
        shift        = 0;
        add          = 0;
        scale        = 0;
        dc_scale     = 0;
    }

    if (lc->cu.pred_mode == MODE_INTER && s->sps->explicit_rdpcm_enabled_flag &&
        (transform_skip_flag || lc->cu.cu_transquant_bypass_flag)) {
        explicit_rdpcm_flag = explicit_rdpcm_flag_decode(s, c_idx);
        if (explicit_rdpcm_flag) {
            explicit_rdpcm_dir_flag = explicit_rdpcm_dir_flag_decode(s, c_idx);
        }
    }

    last_significant_coeff_xy_prefix_decode(s, c_idx, log2_trafo_size,
                                           &last_significant_coeff_x, &last_significant_coeff_y);

    if (last_significant_coeff_x > 3) {
        int suffix = last_significant_coeff_suffix_decode(s, last_significant_coeff_x);
        last_significant_coeff_x = (1 << ((last_significant_coeff_x >> 1) - 1)) *
        (2 + (last_significant_coeff_x & 1)) +
        suffix;
    }

    if (last_significant_coeff_y > 3) {
        int suffix = last_significant_coeff_suffix_decode(s, last_significant_coeff_y);
        last_significant_coeff_y = (1 << ((last_significant_coeff_y >> 1) - 1)) *
        (2 + (last_significant_coeff_y & 1)) +
        suffix;
    }

    if (scan_idx == SCAN_VERT)
        FFSWAP(int, last_significant_coeff_x, last_significant_coeff_y);

    x_cg_last_sig = last_significant_coeff_x >> 2;
    y_cg_last_sig = last_significant_coeff_y >> 2;

    switch (scan_idx) {
    case SCAN_DIAG: {
        int last_x_c = last_significant_coeff_x & 3;
        int last_y_c = last_significant_coeff_y & 3;

        scan_x_off = ff_hevc_diag_scan4x4_x;
        scan_y_off = ff_hevc_diag_scan4x4_y;
        num_coeff = diag_scan4x4_inv[last_y_c][last_x_c];
        if (trafo_size == 4) {
            scan_x_cg = scan_1x1;
            scan_y_cg = scan_1x1;
        } else if (trafo_size == 8) {
            num_coeff += diag_scan2x2_inv[y_cg_last_sig][x_cg_last_sig] << 4;
            scan_x_cg = diag_scan2x2_x;
            scan_y_cg = diag_scan2x2_y;
        } else if (trafo_size == 16) {
            num_coeff += diag_scan4x4_inv[y_cg_last_sig][x_cg_last_sig] << 4;
            scan_x_cg = ff_hevc_diag_scan4x4_x;
```

```c
            scan_x_cg = ff_hevc_diag_scan4x4_x;
            scan_y_cg = ff_hevc_diag_scan4x4_y;
        } else { // trafo_size == 32
            num_coeff += diag_scan8x8_inv[y_cg_last_sig][x_cg_last_sig] << 4;
            scan_x_cg = ff_hevc_diag_scan8x8_x;
            scan_y_cg = ff_hevc_diag_scan8x8_y;
        }
        break;
    }
    case SCAN_HORIZ:
        scan_x_cg = horiz_scan2x2_x;
        scan_y_cg = horiz_scan2x2_y;
        scan_x_off = horiz_scan4x4_x;
        scan_y_off = horiz_scan4x4_y;
        num_coeff = horiz_scan8x8_inv[last_significant_coeff_y][last_significant_coeff_x];
        break;
    default: //SCAN_VERT
        scan_x_cg = horiz_scan2x2_y;
        scan_y_cg = horiz_scan2x2_x;
        scan_x_off = horiz_scan4x4_y;
        scan_y_off = horiz_scan4x4_x;
        num_coeff = horiz_scan8x8_inv[last_significant_coeff_x][last_significant_coeff_y];
        break;
    }
    num_coeff++;
    num_last_subset = (num_coeff - 1) >> 4;

    for (i = num_last_subset; i >= 0; i--) {
        int n, m;
        int x_cg, y_cg, x_c, y_c, pos;
        int implicit_non_zero_coeff = 0;
        int64_t trans_coeff_level;
        int prev_sig = 0;
        int offset = i << 4;
        int rice_init = 0;

        uint8_t significant_coeff_flag_idx[16];
        uint8_t nb_significant_coeff_flag = 0;

        x_cg = scan_x_cg[i];
        y_cg = scan_y_cg[i];

        if ((i < num_last_subset) && (i > 0)) {
            int ctx_cg = 0;
            if (x_cg < (1 << (log2_trafo_size - 2)) - 1)
                ctx_cg += significant_coeff_group_flag[x_cg + 1][y_cg];
            if (y_cg < (1 << (log2_trafo_size - 2)) - 1)
                ctx_cg += significant_coeff_group_flag[x_cg][y_cg + 1];

            significant_coeff_group_flag[x_cg][y_cg] =
                significant_coeff_group_flag_decode(s, c_idx, ctx_cg);
            implicit_non_zero_coeff = 1;
        } else {
            significant_coeff_group_flag[x_cg][y_cg] =
            ((x_cg == x_cg_last_sig && y_cg == y_cg_last_sig) ||
             (x_cg == 0 && y_cg == 0));
        }

        last_scan_pos = num_coeff - offset - 1;

        if (i == num_last_subset) {
            n_end = last_scan_pos - 1;
            significant_coeff_flag_idx[0] = last_scan_pos;
            nb_significant_coeff_flag = 1;
        } else {
            n_end = 15;
        }

        if (x_cg < ((1 << log2_trafo_size) - 1) >> 2)
            prev_sig = !!significant_coeff_group_flag[x_cg + 1][y_cg];
        if (y_cg < ((1 << log2_trafo_size) - 1) >> 2)
            prev_sig += (!!significant_coeff_group_flag[x_cg][y_cg + 1] << 1);

        if (significant_coeff_group_flag[x_cg][y_cg] && n_end >= 0) {
            static const uint8_t ctx_idx_map[] = {
                0, 1, 4, 5, 2, 3, 4, 5, 6, 6, 8, 8, 7, 7, 8, 8, // log2_trafo_size == 2
                1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, // prev_sig == 0
                2, 2, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, // prev_sig == 1
                2, 1, 0, 0, 2, 1, 0, 0, 2, 1, 0, 0, 2, 1, 0, 0, // prev_sig == 2
                2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2  // default
            };
            const uint8_t *ctx_idx_map_p;
            int scf_offset = 0;
            if (s->sps->transform_skip_context_enabled_flag &&
                (transform_skip_flag || lc->cu.cu_transquant_bypass_flag)) {
                ctx_idx_map_p = (uint8_t*) &ctx_idx_map[4 * 16];
                if (c_idx == 0) {
                    scf_offset = 40;
                } else {
                    scf_offset = 14 + 27;
                }
            } else {
```

```
            } else {
                if (c_idx != 0)
                    scf_offset = 27;
                if (log2_trafo_size == 2) {
                    ctx_idx_map_p = (uint8_t*) &ctx_idx_map[0];
                } else {
                    ctx_idx_map_p = (uint8_t*) &ctx_idx_map[(prev_sig + 1) << 4];
                    if (c_idx == 0) {
                        if ((x_cg > 0 || y_cg > 0))
                            scf_offset += 3;
                        if (log2_trafo_size == 3) {
                            scf_offset += (scan_idx == SCAN_DIAG) ? 9 : 15;
                        } else {
                            scf_offset += 21;
                        }
                    } else {
                        if (log2_trafo_size == 3)
                            scf_offset += 9;
                        else
                            scf_offset += 12;
                    }
                }
            }
            for (n = n_end; n > 0; n--) {
                x_c = scan_x_off[n];
                y_c = scan_y_off[n];
                if (significant_coeff_flag_decode(s, x_c, y_c, scf_offset, ctx_idx_map_p)) {
                    significant_coeff_flag_idx[nb_significant_coeff_flag] = n;
                    nb_significant_coeff_flag++;
                    implicit_non_zero_coeff = 0;
                }
            }
            if (implicit_non_zero_coeff == 0) {
                if (s->sps->transform_skip_context_enabled_flag &&
                    (transform_skip_flag || lc->cu.cu_transquant_bypass_flag)) {
                    if (c_idx == 0) {
                        scf_offset = 42;
                    } else {
                        scf_offset = 16 + 27;
                    }
                } else {
                    if (i == 0) {
                        if (c_idx == 0)
                            scf_offset = 0;
                        else
                            scf_offset = 27;
                    } else {
                        scf_offset = 2 + scf_offset;
                    }
                }
                if (significant_coeff_flag_decode_0(s, c_idx, scf_offset) == 1) {
                    significant_coeff_flag_idx[nb_significant_coeff_flag] = 0;
                    nb_significant_coeff_flag++;
                }
            } else {
                significant_coeff_flag_idx[nb_significant_coeff_flag] = 0;
                nb_significant_coeff_flag++;
            }
        }

        n_end = nb_significant_coeff_flag;


        if (n_end) {
            int first_nz_pos_in_cg;
            int last_nz_pos_in_cg;
            int c_rice_param = 0;
            int first_greater1_coeff_idx = -1;
            uint8_t coeff_abs_level_greater1_flag[8];
            uint16_t coeff_sign_flag;
            int sum_abs = 0;
            int sign_hidden;
            int sb_type;


            // initialize first elem of coeff_bas_level_greater1_flag
            int ctx_set = (i > 0 && c_idx == 0) ? 2 : 0;

            if (s->sps->persistent_rice_adaptation_enabled_flag) {
                if (!transform_skip_flag && !lc->cu.cu_transquant_bypass_flag)
                    sb_type = 2 * (c_idx == 0 ? 1 : 0);
                else
                    sb_type = 2 * (c_idx == 0 ? 1 : 0) + 1;
                c_rice_param = lc->stat_coeff[sb_type] / 4;
            }

            if (!(i == num_last_subset) && greater1_ctx == 0)
                ctx_set++;
            greater1_ctx = 1;
            last_nz_pos_in_cg = significant_coeff_flag_idx[0];

            for (m = 0; m < (n_end > 8 ? 8 : n_end); m++) {
```

```
                int inc = (ctx_set << 2) + greater1_ctx;
                coeff_abs_level_greater1_flag[m] =
                    coeff_abs_level_greater1_flag_decode(s, c_idx, inc);
                if (coeff_abs_level_greater1_flag[m]) {
                    greater1_ctx = 0;
                    if (first_greater1_coeff_idx == -1)
                        first_greater1_coeff_idx = m;
                } else if (greater1_ctx > 0 && greater1_ctx < 3) {
                    greater1_ctx++;
                }
            }
            first_nz_pos_in_cg = significant_coeff_flag_idx[n_end - 1];

            if (lc->cu.cu_transquant_bypass_flag ||
                (lc->cu.pred_mode ==  MODE_INTRA  &&
                 s->sps->implicit_rdpcm_enabled_flag  &&  transform_skip_flag  &&
                 (pred_mode_intra == 10 || pred_mode_intra  ==  26 )) ||
                 explicit_rdpcm_flag)
                sign_hidden = 0;
            else
                sign_hidden = (last_nz_pos_in_cg - first_nz_pos_in_cg >= 4);

            if (first_greater1_coeff_idx != -1) {
                coeff_abs_level_greater1_flag[first_greater1_coeff_idx] += coeff_abs_level_greater2_flag_decode(s, c_idx, ctx_set);
            }
            if (!s->pps->sign_data_hiding_flag || !sign_hidden ) {
                coeff_sign_flag = coeff_sign_flag_decode(s, nb_significant_coeff_flag) << (16 - nb_significant_coeff_flag);
            } else {
                coeff_sign_flag = coeff_sign_flag_decode(s, nb_significant_coeff_flag - 1) << (16 - (nb_significant_coeff_flag - 1));
            }

            for (m = 0; m < n_end; m++) {
                n = significant_coeff_flag_idx[m];
                GET_COORD(offset, n);
                if (m < 8) {
                    trans_coeff_level = 1 + coeff_abs_level_greater1_flag[m];
                    if (trans_coeff_level == ((m == first_greater1_coeff_idx) ? 3 : 2)) {
                        int last_coeff_abs_level_remaining = coeff_abs_level_remaining_decode(s, c_rice_param);

                        trans_coeff_level += last_coeff_abs_level_remaining;
                        if (trans_coeff_level > (3 << c_rice_param))
                            c_rice_param = s->sps->persistent_rice_adaptation_enabled_flag ? c_rice_param + 1 : FFMIN(c_rice_param + 1, 4);
                        if (s->sps->persistent_rice_adaptation_enabled_flag && !rice_init) {
                            int c_rice_p_init = lc->stat_coeff[sb_type] / 4;
                            if (last_coeff_abs_level_remaining >= (3 << c_rice_p_init))
                                lc->stat_coeff[sb_type]++;
                            else if (2 * last_coeff_abs_level_remaining < (1 << c_rice_p_init))
                                if (lc->stat_coeff[sb_type] > 0)
                                    lc->stat_coeff[sb_type]--;
                            rice_init = 1;
                        }
                    }
                } else {
                    int last_coeff_abs_level_remaining = coeff_abs_level_remaining_decode(s, c_rice_param);

                    trans_coeff_level = 1 + last_coeff_abs_level_remaining;
                    if (trans_coeff_level > (3 << c_rice_param))
                        c_rice_param = s->sps->persistent_rice_adaptation_enabled_flag ? c_rice_param + 1 : FFMIN(c_rice_param + 1, 4);
                    if (s->sps->persistent_rice_adaptation_enabled_flag && !rice_init) {
                        int c_rice_p_init = lc->stat_coeff[sb_type] / 4;
                        if (last_coeff_abs_level_remaining >= (3 << c_rice_p_init))
                            lc->stat_coeff[sb_type]++;
                        else if (2 * last_coeff_abs_level_remaining < (1 << c_rice_p_init))
                            if (lc->stat_coeff[sb_type] > 0)
                                lc->stat_coeff[sb_type]--;
                        rice_init = 1;
                    }
                }
                if (s->pps->sign_data_hiding_flag && sign_hidden) {
                    sum_abs += trans_coeff_level;
                    if (n == first_nz_pos_in_cg && (sum_abs&1))
                        trans_coeff_level = -trans_coeff_level;
                }
                if (coeff_sign_flag >> 15)
                    trans_coeff_level = -trans_coeff_level;
                coeff_sign_flag <<= 1;
                if(!lc->cu.cu_transquant_bypass_flag) {
                    if (s->sps->scaling_list_enable_flag && !(transform_skip_flag && log2_trafo_size > 2)) {
                        if(y_c || x_c || log2_trafo_size < 4) {
                            switch(log2_trafo_size) {
                                case 3: pos = (y_c << 3) + x_c; break;
                                case 4: pos = ((y_c >> 1) << 3) + (x_c >> 1); break;
                                case 5: pos = ((y_c >> 2) << 3) + (x_c >> 2); break;
                                default: pos = (y_c << 2) + x_c; break;
                            }
                            scale_m = scale_matrix[pos];
                        } else {
                            scale_m = dc_scale;
                        }
```

```
                    }
                    trans_coeff_level = (trans_coeff_level * (int64_t)scale * (int64_t)scale_m + add) >> shift;
                    if(trans_coeff_level < 0) {
                        if((~trans_coeff_level) & 0xFffffffffff8000)
                            trans_coeff_level = -32768;
                    } else {
                        if(trans_coeff_level & 0xffffffffffff8000)
                            trans_coeff_level = 32767;
                    }
                }
                coeffs[y_c * trafo_size + x_c] = trans_coeff_level;
            }
        }
    }

    if (lc->cu.cu_transquant_bypass_flag) {
        if (explicit_rdpcm_flag || (s->sps->implicit_rdpcm_enabled_flag &&
                                    (pred_mode_intra == 10 || pred_mode_intra == 26))) {
            int mode = s->sps->implicit_rdpcm_enabled_flag ? (pred_mode_intra == 26) : explicit_rdpcm_dir_flag;

            s->hevcdsp.transform_rdpcm(coeffs, log2_trafo_size, mode);
        }
    } else {
        if (transform_skip_flag) {
            int rot = s->sps->transform_skip_rotation_enabled_flag &&
                      log2_trafo_size == 2 &&
                      lc->cu.pred_mode == MODE_INTRA;
            if (rot) {
                for (i = 0; i < 8; i++)
                    FFSWAP(int16_t, coeffs[i], coeffs[16 - i - 1]);
            }

            s->hevcdsp.transform_skip(coeffs, log2_trafo_size);

            if (explicit_rdpcm_flag || (s->sps->implicit_rdpcm_enabled_flag &&
                                        lc->cu.pred_mode == MODE_INTRA &&
                                        (pred_mode_intra == 10 || pred_mode_intra == 26))) {
                int mode = explicit_rdpcm_flag ? explicit_rdpcm_dir_flag : (pred_mode_intra == 26);

                s->hevcdsp.transform_rdpcm(coeffs, log2_trafo_size, mode);
            }
        } else if (lc->cu.pred_mode == MODE_INTRA && c_idx == 0 && log2_trafo_size == 2) {
            //这里是4x4DST
            s->hevcdsp.idct_4x4_luma(coeffs);
        } else {
            int max_xy = FFMAX(last_significant_coeff_x, last_significant_coeff_y);
            if (max_xy == 0)
                s->hevcdsp.idct_dc[log2_trafo_size-2](coeffs);//只对DC系数做IDCT的比较快的算法
            else {
                int col_limit = last_significant_coeff_x + last_significant_coeff_y + 4;
                if (max_xy < 4)
                    col_limit = FFMIN(4, col_limit);
                else if (max_xy < 8)
                    col_limit = FFMIN(8, col_limit);
                else if (max_xy < 12)
                    col_limit = FFMIN(24, col_limit);
                s->hevcdsp.idct[log2_trafo_size-2](coeffs, col_limit);//普通的IDCT
            }
        }
    }
    if (lc->tu.cross_pf) {
        int16_t *coeffs_y = (int16_t*)lc->edge_emu_buffer;

        for (i = 0; i < (trafo_size * trafo_size); i++) {
            coeffs[i] = coeffs[i] + ((lc->tu.res_scale_val * coeffs_y[i]) >> 3);
        }
    }
    //将IDCT的结果叠加到预测数据上
    s->hevcdsp.transform_add[log2_trafo_size-2](dst, coeffs, stride);
}
```

ff_hevc_hls_residual_coding()前半部分的一大段代码应该是用于解析残差数据的（目前还没有细看），后半部分的代码则用于对残差数据进行DCT变换。在DCT反变换的时候，调用了如下几种功能的汇编函数：

HEVCDSPContext-> idct_4x4_luma()：4x4DST反变换

HEVCDSPContext-> idct_dc[X]()：特殊的只包含DC系数的DCT反变换

HEVCDSPContext-> idct[X]()：普通的DCT反变换

HEVCDSPContext-> transform_add [X]()：残差像素数据叠加

其中不同的[X]取值代表了不同尺寸的系数块：

[0]代表4x4；

[1]代表8x8；

[2]代表16x16；

[3]代表32x32；

后文将会对上述汇编函数进行详细分析。

帧内预测和DCT反变换知识

HEVC标准中的帧内预测和DCT反变换都是以TU为单位进行的，因此将这两部分知识放到一起记录。

帧内预测知识

HEVC的帧内预测共有35中预测模式，如下表所示：

模式编号	模式名称
0	Planar
1	DC
2-34	33种角度预测模式

其中第2-34种预测方式的角度如下所示。

HEVC的角度预测方向相对于H.264增加到了33种。这样做的好处是能够更有效低表示图像的纹理特征，提高预测精度。其中编号2到17的角度预测模式为水平类模式，编号为18到34的角度预测模式为垂直类模式。编号为10的为水平预测，编号为26的位垂直预测模式。
Planar模式的计算方式如下图所示。

从图中可以看出，Planar模式首先将左边一列像素最下面一个像素值水平复制一行，将上边一行像素最右边一个像素值垂直复制一列；然后使用类似于双线性插值的方式，获得预测数据。这一预测方式综合了水平和垂直预测的特点。
DC模式的计算方法如下图所示。

从图中可以看出，DC模式计算方式原理很简单：直接将当前块上方一行以及左边一列像素求得平均值后，赋值给当前块中的每一个像素。

DCT变换

H.264中采用了4x4整数DCT变换，在HEVC中沿用了这种整数变换方法，但是其主要有以下几点不同：

（1）变换尺寸不再限于4x4，而是包括了4x4，8x8，16x16，32x32几种方式。

（2）变换系数值变大了很多，这样使得整数DCT的结果更接近浮点DCT的结果。注意在变换完成后会乘以修正矩阵（对于4x4变换来说，统一乘以1/128；对于尺寸N，修正系数值为1/(64*sqrt(N))）将放大后的结果修正回来。

（3）在Intra4x4亮度残差变换的时候使用了一种比较特殊的4x4DST（离散正弦变换，中间的“S”代表“sin()”），在后文会记录该种变换。

HEVC支持最大为32x32的DCT变换。该变换矩阵的系数值如下图所示。其中第一张图为左边的16列数值，第二张图为右边的16列数值。

4x4DCT变换的系数来自于为32x32系数矩阵中第0，8，16，24行元素中的前4个元素，在图中以红色方框表示出来。由此可知4x4DCT系数矩阵为：

64 64 64 64
83 36 -36 -83
64 -64 -64 64
36 -83 83 -36

8x8DCT变换的系数来自于32x32系数矩阵中第0，4，8，12，16，20，24，28行元素中的前8个元素，在图中以黄色方框表示出来。由此可知8x8 DCT系数矩阵为：

64 64 64 64 64 64 64 64
89 75 50 18 -18 -50 -75 -89
83 36 -36 -83 -83 -36 36 83
75 -18 -89 -50 50 89 18 -75
64 -64 -64 64 64 -64 -64 64
50 -89 18 75 -75 -18 89 -50
36 -83 83 -36 -36 83 -83 36
18 -50 75 -89 89 -75 50 -18

16x16　DCT变换的系数来自于32x32系数矩阵中第0，2，4…，28，30行元素中的前16个元素，在图中以绿色方框表示出来。由于系数数量较大，就不再列出了。

在编码Intra4x4的残差数据的时候，使用了一种比较特殊的4x4DST。该种变换的系数矩阵如下所示。相关的实验表明，在编码Intra4x4的时候使用4x4DST可以提升约0.8%的编码效率。

29 55 74 84
74 74 0 -74
84 -29 -74 55
55 -84 74 -29

帧内预测实例

本节以一小段视频的码流为例，看一下HEVC码流中的帧内预测相关的信息。
【示例1】
下图为一个I帧解码后的图像。

下图为该帧CTU的划分方式。可以看出画面复杂的地方CTU划分比较细。

下图的蓝色线条显示了帧内预测的方向。

下图显示了帧内预测方向与图像内容之间的关系。可以看出帧内预测方向基本上和图像纹理方向是一致的。

下图为经过帧内预测，没有经过残差叠加处理的视频内容。

下图为该帧的残差信息。

【示例2】
下图为一个I帧解码后的图像。

下图为该帧CTU的划分方式。

下图的蓝色线条显示了帧内预测的方向。

下图显示了帧内预测方向与图像内容之间的关系。

下图为经过帧内预测，没有经过残差叠加处理的视频内容。

下图为该帧的残差信息。

【示例3-帧内滤波信息】
本节以一段《Sintel》动画的码流为例，看一下HEVC码流中的帧内滤波具体的信息。下图为I帧解码后的图像。

下图为没有叠加残差数据的帧内预测的结果。在这里我们选择一个8x8 CU（图中以紫色方框标出）看一下其中具体的信息。该CU采用了19号帧内预测模式（属于角度Angular预测模式）。

该8x8 CU的帧内预测信息如下图所示。

【示例4-DCT反变换示例】
本节还是以《Sintel》动画的码流为例，看一下HEVC码流中的DCT反变换具体的信息。下图为一帧解码后的图像。

下图为该帧图像的残差数据。在这里我们选择一个8x8 CU（图中以紫色方框标出）看一下其中具体的信息。

该8x8 CU的DCT反变换信息如下图所示。图中显示了反量化，反变换的具体过程。

帧内预测汇编函数源代码

帧内预测相关的汇编函数位于HEVCPredContext中。HEVCPredContext的初始化函数是ff_hevc_pred_init()。该函数对HEVCPredContext结构体中的函数指针进行了赋值。FFmpeg HEVC解码器运行的过程中只要调用HEVCPredContext的函数指针就可以完成相应的功能。

ff_hevc_pred_init()

ff_hevc_pred_init()用于初始化HEVCPredContext结构体中的汇编函数指针。该函数的定义如下所示。

[cpp]

```cpp
//帧内预测函数初始化
void ff_hevc_pred_init(HEVCPredContext *hpc, int bit_depth)
{
#undef FUNC
#define FUNC(a, depth) a ## _ ## depth

#define HEVC_PRED(depth)                                \
    hpc->intra_pred[0]   = FUNC(intra_pred_2, depth);   \
    hpc->intra_pred[1]   = FUNC(intra_pred_3, depth);   \
    hpc->intra_pred[2]   = FUNC(intra_pred_4, depth);   \
    hpc->intra_pred[3]   = FUNC(intra_pred_5, depth);   \
    hpc->pred_planar[0]  = FUNC(pred_planar_0, depth);  \
    hpc->pred_planar[1]  = FUNC(pred_planar_1, depth);  \
    hpc->pred_planar[2]  = FUNC(pred_planar_2, depth);  \
    hpc->pred_planar[3]  = FUNC(pred_planar_3, depth);  \
    hpc->pred_dc         = FUNC(pred_dc, depth);        \
    hpc->pred_angular[0] = FUNC(pred_angular_0, depth); \
    hpc->pred_angular[1] = FUNC(pred_angular_1, depth); \
    hpc->pred_angular[2] = FUNC(pred_angular_2, depth); \
    hpc->pred_angular[3] = FUNC(pred_angular_3, depth);

    switch (bit_depth) {
    case 9:
        HEVC_PRED(9);
        break;
    case 10:
        HEVC_PRED(10);
        break;
    case 12:
        HEVC_PRED(12);
        break;
    default:
        HEVC_PRED(8);
        break;
    }
}
```

从源代码可以看出，ff_hevc_pred_init()函数中包含一个名为“HEVC_PRED(depth)”的很长的宏定义。该宏定义中包含了C语言版本的帧内预测函数的初始化代码。ff_hevc_dsp_init()会根据系统的颜色位深bit_depth初始化相应的C语言版本的帧内预测函数。下面以8bit颜色位深为例，看一下“HEVC_ PRED(8)”的展开结果。

[cpp]

```cpp
hpc->intra_pred[0]   = intra_pred_2_8;
hpc->intra_pred[1]   = intra_pred_3_8;
hpc->intra_pred[2]   = intra_pred_4_8;
hpc->intra_pred[3]   = intra_pred_5_8;
hpc->pred_planar[0]  = pred_planar_0_8;
hpc->pred_planar[1]  = pred_planar_1_8;
hpc->pred_planar[2]  = pred_planar_2_8;
hpc->pred_planar[3]  = pred_planar_3_8;
hpc->pred_dc         = pred_dc_8;
hpc->pred_angular[0] = pred_angular_0_8;
hpc->pred_angular[1] = pred_angular_1_8;
hpc->pred_angular[2] = pred_angular_2_8;
hpc->pred_angular[3] = pred_angular_3_8;
```

可以看出“HEVC_ PRED(8)”初始化了帧内预测模块的C语言版本函数。HEVCPredContext的定义如下。

[cpp]

```cpp
typedef struct HEVCPredContext {
    void (*intra_pred[4])(struct HEVCContext *s, int x0, int y0, int c_idx);

    void (*pred_planar[4])(uint8_t *src, const uint8_t *top,
                           const uint8_t *left, ptrdiff_t stride);
    void (*pred_dc)(uint8_t *src, const uint8_t *top, const uint8_t *left,
                    ptrdiff_t stride, int log2_size, int c_idx);
    void (*pred_angular[4])(uint8_t *src, const uint8_t *top,
                            const uint8_t *left, ptrdiff_t stride,
                            int c_idx, int mode);
} HEVCPredContext;
```

从源代码中可以看出，HEVCPredContext中存储了4个汇编函数指针（数组）：

intra_pred[4]()：帧内预测的入口函数，该函数执行过程中调用了后面3个函数指针。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

pred_planar[4]()：Planar预测模式函数。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

pred_dc()：DC预测模式函数。

pred_angular[4]()：角度预测模式。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

下文按照顺序分别介绍这几种函数。

HEVCPredContext ->intra_pred[4]()

intra_pred[4]()是帧内预测的入口函数，该函数执行过程中调用了Planar、DC或者角度预测函数。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块

的具体的处理函数为：

- intra_pred_2_8()——4x4块
- intra_pred_3_8()——8x8块
- intra_pred_4_8()——16x16块
- intra_pred_5_8()——32x32块

PS：函数命名时候中间的数字是块的边长取log2()之后的数值。

上面这几个函数的定义如下所示。

[cpp]

```cpp
#define INTRA_PRED(size)                                                        \
static void FUNC(intra_pred_ ## size)(HEVCContext *s, int x0, int y0, int c_idx)    \
{                                                                               \
    FUNC(intra_pred)(s, x0, y0, size, c_idx);                                   \
}

/* 几种不同大小的方块对应的帧内预测函数
 * 参数是方块像素数取对数之后的值
 * 例如“INTRA_PRED(2)”即为4x4块的帧内预测函数
 *
 * “INTRA_PRED(2)”展开后的函数是
 * static void intra_pred_2_8(HEVCContext *s, int x0, int y0, int c_idx)
 * {
 *  intra_pred_8(s, x0, y0, 2, c_idx);
 * }
 */
INTRA_PRED(2)
INTRA_PRED(3)
INTRA_PRED(4)
INTRA_PRED(5)
```

从源代码中可以看出，intra_pred_2_8()、intra_pred_3_8()等函数都是通过“INTRA_PRED()”宏进行定义的。intra_pred_2_8()、intra_pred_3_8()的函数的内部都调用了同一个函数intra_pred_8()。这几个函数唯一的不同在于，调用intra_pred_8()时候第4个参数size的值不一样。

intra_pred_8()

intra_pred_8()完成了帧内预测前的滤波等准备工作，并根据帧内预测类型的不同（Planar、DC、角度）调用不同的帧内预测函数。该函数的定义如下所示。

[cpp]

```cpp
static av_always_inline void FUNC(intra_pred)(HEVCContext *s, int x0, int y0,
                                              int log2_size, int c_idx)
{
#define PU(x) \
    ((x) >> s->sps->log2_min_pu_size)
#define MVF(x, y) \
    (s->ref->tab_mvf[(x) + (y) * min_pu_width])
#define MVF_PU(x, y) \
    MVF(PU(x0 + ((x) << hshift)), PU(y0 + ((y) << vshift)))
#define IS_INTRA(x, y) \
    (MVF_PU(x, y).pred_flag == PF_INTRA)
#define MIN_TB_ADDR_ZS(x, y) \
    s->pps->min_tb_addr_zs[(y) * (s->sps->tb_mask+2) + (x)]
#define EXTEND(ptr, val, len)         \
do {                                  \
    pixel4 pix = PIXEL_SPLAT_X4(val); \
    for (i = 0; i < (len); i += 4)    \
        AV_WN4P(ptr + i, pix);        \
} while (0)

#define EXTEND_RIGHT_CIP(ptr, start, length)                                   \
        for (i = start; i < (start) + (length); i += 4)                        \
            if (!IS_INTRA(i, -1))                                              \
                AV_WN4P(&ptr[i], a);                                           \
            else                                                               \
                a = PIXEL_SPLAT_X4(ptr[i+3])
#define EXTEND_LEFT_CIP(ptr, start, length) \
        for (i = start; i > (start) - (length); i--) \
            if (!IS_INTRA(i - 1, -1)) \
                ptr[i - 1] = ptr[i]
#define EXTEND_UP_CIP(ptr, start, length)                                      \
        for (i = (start); i > (start) - (length); i -= 4)                      \
            if (!IS_INTRA(-1, i - 3))                                          \
                AV_WN4P(&ptr[i - 3], a);                                       \
            else                                                               \
                a = PIXEL_SPLAT_X4(ptr[i - 3])
#define EXTEND_DOWN_CIP(ptr, start, length)                                    \
        for (i = start; i < (start) + (length); i += 4)                        \
            if (!IS_INTRA(-1, i))                                              \
                AV_WN4P(&ptr[i], a);                                           \
            else                                                               \
                a = PIXEL_SPLAT_X4(ptr[i + 3])

    HEVCLocalContext *lc = s->HEVClc;
    int i;
    int hshift = s->sps->hshift[c_idx];
    int vshift = s->sps->vshift[c_idx];
    int size = (1 << log2_size);
    int size_in_luma_h = size << hshift;
```

```c
    int size_in_tbs_h  = size_in_luma_h >> s->sps->log2_min_tb_size;
    int size_in_luma_v = size << vshift;
    int size_in_tbs_v  = size_in_luma_v >> s->sps->log2_min_tb_size;
    int x = x0 >> hshift;
    int y = y0 >> vshift;
    int x_tb = (x0 >> s->sps->log2_min_tb_size) & s->sps->tb_mask;
    int y_tb = (y0 >> s->sps->log2_min_tb_size) & s->sps->tb_mask;

    int cur_tb_addr = MIN_TB_ADDR_ZS(x_tb, y_tb);
    //注意c_idx标志了颜色分量
    ptrdiff_t stride = s->frame->linesize[c_idx] / sizeof(pixel);
    pixel *src = (pixel*)s->frame->data[c_idx] + x + y * stride;

    int min_pu_width = s->sps->min_pu_width;

    enum IntraPredMode mode = c_idx ? lc->tu.intra_pred_mode_c :
                              lc->tu.intra_pred_mode;
    pixel4 a;
    pixel  left_array[2 * MAX_TB_SIZE + 1];
    pixel  filtered_left_array[2 * MAX_TB_SIZE + 1];
    pixel  top_array[2 * MAX_TB_SIZE + 1];
    pixel  filtered_top_array[2 * MAX_TB_SIZE + 1];

    pixel  *left          = left_array + 1;
    pixel  *top           = top_array  + 1;
    pixel  *filtered_left = filtered_left_array + 1;
    pixel  *filtered_top  = filtered_top_array  + 1;
    int cand_bottom_left = lc->na.cand_bottom_left && cur_tb_addr > MIN_TB_ADDR_ZS( x_tb - 1, (y_tb + size_in_tbs_v) & s->sps->tb_mask);
    int cand_left        = lc->na.cand_left;
    int cand_up_left     = lc->na.cand_up_left;
    int cand_up          = lc->na.cand_up;
    int cand_up_right    = lc->na.cand_up_right    && cur_tb_addr > MIN_TB_ADDR_ZS((x_tb + size_in_tbs_h) & s->sps->tb_mask, y_tb - 1);

    int bottom_left_size = (FFMIN(y0 + 2 * size_in_luma_v, s->sps->height) -
                           (y0 + size_in_luma_v)) >> vshift;
    int top_right_size   = (FFMIN(x0 + 2 * size_in_luma_h, s->sps->width) -
                           (x0 + size_in_luma_h)) >> hshift;

    if (s->pps->constrained_intra_pred_flag == 1) {
        int size_in_luma_pu_v = PU(size_in_luma_v);
        int size_in_luma_pu_h = PU(size_in_luma_h);
        int on_pu_edge_x    = !(x0 & ((1 << s->sps->log2_min_pu_size) - 1));
        int on_pu_edge_y    = !(y0 & ((1 << s->sps->log2_min_pu_size) - 1));
        if (!size_in_luma_pu_h)
            size_in_luma_pu_h++;
        if (cand_bottom_left == 1 && on_pu_edge_x) {
            int x_left_pu   = PU(x0 - 1);
            int y_bottom_pu = PU(y0 + size_in_luma_v);
            int max = FFMIN(size_in_luma_pu_v, s->sps->min_pu_height - y_bottom_pu);
            cand_bottom_left = 0;
            for (i = 0; i < max; i += 2)
                cand_bottom_left |= (MVF(x_left_pu, y_bottom_pu + i).pred_flag == PF_INTRA);
        }
        if (cand_left == 1 && on_pu_edge_x) {
            int x_left_pu   = PU(x0 - 1);
            int y_left_pu   = PU(y0);
            int max = FFMIN(size_in_luma_pu_v, s->sps->min_pu_height - y_left_pu);
            cand_left = 0;
            for (i = 0; i < max; i += 2)
                cand_left |= (MVF(x_left_pu, y_left_pu + i).pred_flag == PF_INTRA);
        }
        if (cand_up_left == 1) {
            int x_left_pu   = PU(x0 - 1);
            int y_top_pu    = PU(y0 - 1);
            cand_up_left = MVF(x_left_pu, y_top_pu).pred_flag == PF_INTRA;
        }
        if (cand_up == 1 && on_pu_edge_y) {
            int x_top_pu    = PU(x0);
            int y_top_pu    = PU(y0 - 1);
            int max = FFMIN(size_in_luma_pu_h, s->sps->min_pu_width - x_top_pu);
            cand_up = 0;
            for (i = 0; i < max; i += 2)
                cand_up |= (MVF(x_top_pu + i, y_top_pu).pred_flag == PF_INTRA);
        }
        if (cand_up_right == 1 && on_pu_edge_y) {
            int y_top_pu    = PU(y0 - 1);
            int x_right_pu  = PU(x0 + size_in_luma_h);
            int max = FFMIN(size_in_luma_pu_h, s->sps->min_pu_width - x_right_pu);
            cand_up_right = 0;
            for (i = 0; i < max; i += 2)
                cand_up_right |= (MVF(x_right_pu + i, y_top_pu).pred_flag == PF_INTRA);
        }
        memset(left, 128, 2 * MAX_TB_SIZE*sizeof(pixel));
        memset(top , 128, 2 * MAX_TB_SIZE*sizeof(pixel));
        top[-1] = 128;
    }
    if (cand_up_left) {
        left[-1] = POS(-1, -1);
        top[-1]  = left[-1];
    }
```

```
    }
    if (cand_up)
        memcpy(top, src - stride, size * sizeof(pixel));
    if (cand_up_right) {
        memcpy(top + size, src - stride + size, size * sizeof(pixel));
        EXTEND(top + size + top_right_size, POS(size + top_right_size - 1, -1),
               size - top_right_size);
    }
    if (cand_left)
        for (i = 0; i < size; i++)
            left[i] = POS(-1, i);
    if (cand_bottom_left) {
        for (i = size; i < size + bottom_left_size; i++)
            left[i] = POS(-1, i);
        EXTEND(left + size + bottom_left_size, POS(-1, size + bottom_left_size - 1),
               size - bottom_left_size);
    }

    if (s->pps->constrained_intra_pred_flag == 1) {
        if (cand_bottom_left || cand_left || cand_up_left || cand_up || cand_up_right) {
            int size_max_x = x0 + ((2 * size) << hshift) < s->sps->width ?
                                    2 * size : (s->sps->width - x0) >> hshift;
            int size_max_y = y0 + ((2 * size) << vshift) < s->sps->height ?
                                    2 * size : (s->sps->height - y0) >> vshift;
            int j = size + (cand_bottom_left? bottom_left_size: 0) -1;
            if (!cand_up_right) {
                size_max_x = x0 + ((size) << hshift) < s->sps->width ?
                                                    size : (s->sps->width - x0) >> hshift;
            }
            if (!cand_bottom_left) {
                size_max_y = y0 + (( size) << vshift) < s->sps->height ?
                                                     size : (s->sps->height - y0) >> vshift;
            }
            if (cand_bottom_left || cand_left || cand_up_left) {
                while (j > -1 && !IS_INTRA(-1, j))
                    j--;
                if (!IS_INTRA(-1, j)) {
                    j = 0;
                    while (j < size_max_x && !IS_INTRA(j, -1))
                        j++;
                    EXTEND_LEFT_CIP(top, j, j + 1);
                    left[-1] = top[-1];
                }
            } else {
                j = 0;
                while (j < size_max_x && !IS_INTRA(j, -1))
                    j++;
                if (j > 0)
                    if (x0 > 0) {
                        EXTEND_LEFT_CIP(top, j, j + 1);
                    } else {
                        EXTEND_LEFT_CIP(top, j, j);
                        top[-1] = top[0];
                    }
                left[-1] = top[-1];
            }
            left[-1] = top[-1];
            if (cand_bottom_left || cand_left) {
                a = PIXEL_SPLAT_X4(left[-1]);
                EXTEND_DOWN_CIP(left, 0, size_max_y);
            }
            if (!cand_left)
                EXTEND(left, left[-1], size);
            if (!cand_bottom_left)
                EXTEND(left + size, left[size - 1], size);
            if (x0 != 0 && y0 != 0) {
                a = PIXEL_SPLAT_X4(left[size_max_y - 1]);
                EXTEND_UP_CIP(left, size_max_y - 1, size_max_y);
                if (!IS_INTRA(-1, - 1))
                    left[-1] = left[0];
            } else if (x0 == 0) {
                EXTEND(left, 0, size_max_y);
            } else {
                a = PIXEL_SPLAT_X4(left[size_max_y - 1]);
                EXTEND_UP_CIP(left, size_max_y - 1, size_max_y);
            }
            top[-1] = left[-1];
            if (y0 != 0) {
                a = PIXEL_SPLAT_X4(left[-1]);
                EXTEND_RIGHT_CIP(top, 0, size_max_x);
            }
        }
    }
    // Infer the unavailable samples
    if (!cand_bottom_left) {
        if (cand_left) {
            EXTEND(left + size, left[size - 1], size);
        } else if (cand_up_left) {
            EXTEND(left, left[-1], 2 * size);
            cand_left = 1;
        } else if (cand_up) {
            left[-1] = top[0];
```

```
            left[-1] = top[0];
            EXTEND(left, left[-1], 2 * size);
            cand_up_left = 1;
            cand_left    = 1;
        } else if (cand_up_right) {
            EXTEND(top, top[size], size);
            left[-1] = top[size];
            EXTEND(left, left[-1], 2 * size);
            cand_up      = 1;
            cand_up_left = 1;
            cand_left    = 1;
        } else { // No samples available
            left[-1] = (1 << (BIT_DEPTH - 1));
            EXTEND(top,  left[-1], 2 * size);
            EXTEND(left, left[-1], 2 * size);
        }
    }

    if (!cand_left)
        EXTEND(left, left[size], size);
    if (!cand_up_left) {
        left[-1] = left[0];
    }
    if (!cand_up)
        EXTEND(top, left[-1], size);
    if (!cand_up_right)
        EXTEND(top + size, top[size - 1], size);

    top[-1] = left[-1];

    // Filtering process
    // 滤波
    if (!s->sps->intra_smoothing_disabled_flag && (c_idx == 0  || s->sps->chroma_format_idc == 3)) {
        if (mode != INTRA_DC && size != 4){
            int intra_hor_ver_dist_thresh[] = { 7, 1, 0 };
            int min_dist_vert_hor = FFMIN(FFABS((int)(mode - 26U)),
                                          FFABS((int)(mode - 10U)));
            if (min_dist_vert_hor > intra_hor_ver_dist_thresh[log2_size - 3]) {
                int threshold = 1 << (BIT_DEPTH - 5);
                if (s->sps->sps_strong_intra_smoothing_enable_flag && c_idx == 0 &&
                    log2_size == 5 &&
                    FFABS(top[-1]  + top[63]  - 2 * top[31])  < threshold &&
                    FFABS(left[-1] + left[63] - 2 * left[31]) < threshold) {
                    // We can't just overwrite values in top because it could be
                    // a pointer into src
                    filtered_top[-1] = top[-1];
                    filtered_top[63] = top[63];
                    for (i = 0; i < 63; i++)
                        filtered_top[i] = ((64 - (i + 1)) * top[-1] +
                                           (i + 1)  * top[63] + 32) >> 6;
                    for (i = 0; i < 63; i++)
                        left[i] = ((64 - (i + 1)) * left[-1] +
                                   (i + 1)  * left[63] + 32) >> 6;
                    top = filtered_top;
                } else {
                    filtered_left[2 * size - 1] = left[2 * size - 1];
                    filtered_top[2 * size - 1]  = top[2 * size - 1];
                    for (i = 2 * size - 2; i >= 0; i--)
                        filtered_left[i] = (left[i + 1] + 2 * left[i] +
                                            left[i - 1] + 2) >> 2;
                    filtered_top[-1]  =
                    filtered_left[-1] = (left[0] + 2 * left[-1] + top[0] + 2) >> 2;
                    for (i = 2 * size - 2; i >= 0; i--)
                        filtered_top[i] = (top[i + 1] + 2 * top[i] +
                                           top[i - 1] + 2) >> 2;
                    left = filtered_left;
                    top  = filtered_top;
                }
            }
        }
    }
    /*
     * 根据不同的帧内预测模式，调用不同的处理函数
     * pred_planar[4],pred_angular[4]中的“[4]”代表了几种不同大小的方块
     *    [0]:4x4块
     *    [1]:8x8块
     *    [2]:16x16块
     *    [3]:32x32块
     *
     * log2size为方块边长取对数。
     * 4x4块，log2size=log2(4)=2
     * 8x8块，log2size=log2(8)=3
     * 16x16块，log2size=log2(16)=4
     * 32x32块，log2size=log2(32)=5
     *
     */
    switch (mode) {
    case INTRA_PLANAR:
        s->hpc.pred_planar[log2_size - 2]((uint8_t *)src, (uint8_t *)top,
                                          (uint8_t *)left, stride);
        break;
    case INTRA_DC:
```

```
    case INTRA_DC:
        s->hpc.pred_dc((uint8_t *)src, (uint8_t *)top,
                       (uint8_t *)left, stride, log2_size, c_idx);
        break;
    default:
        s->hpc.pred_angular[log2_size - 2]((uint8_t *)src, (uint8_t *)top,
                                           (uint8_t *)left, stride, c_idx,
                                           mode);
        break;
    }
}
```

intra_pred_8()前面部分的代码还没有细看，大致做了一些帧内预测的准备工作；它的后面有一个switch()语句，根据帧内预测模式的不同作不同的处理：

（1）Planar模式，调用HEVCContext-> pred_planar()

（2）DC模式，调用HEVCContext-> pred_dc()

（3）其他模式（剩余都是角度模式），调用HEVCContext-> pred_angular()

HEVC解码器中帧内预测模式的定义于IntraPredMode变量，如下所示。

```cpp
enum IntraPredMode {
    INTRA_PLANAR = 0,
    INTRA_DC,
    INTRA_ANGULAR_2,
    INTRA_ANGULAR_3,
    INTRA_ANGULAR_4,
    INTRA_ANGULAR_5,
    INTRA_ANGULAR_6,
    INTRA_ANGULAR_7,
    INTRA_ANGULAR_8,
    INTRA_ANGULAR_9,
    INTRA_ANGULAR_10,
    INTRA_ANGULAR_11,
    INTRA_ANGULAR_12,
    INTRA_ANGULAR_13,
    INTRA_ANGULAR_14,
    INTRA_ANGULAR_15,
    INTRA_ANGULAR_16,
    INTRA_ANGULAR_17,
    INTRA_ANGULAR_18,
    INTRA_ANGULAR_19,
    INTRA_ANGULAR_20,
    INTRA_ANGULAR_21,
    INTRA_ANGULAR_22,
    INTRA_ANGULAR_23,
    INTRA_ANGULAR_24,
    INTRA_ANGULAR_25,
    INTRA_ANGULAR_26,
    INTRA_ANGULAR_27,
    INTRA_ANGULAR_28,
    INTRA_ANGULAR_29,
    INTRA_ANGULAR_30,
    INTRA_ANGULAR_31,
    INTRA_ANGULAR_32,
    INTRA_ANGULAR_33,
    INTRA_ANGULAR_34,
};
```

下面分别看一下3种帧内预测函数。

HEVCPredContext -> pred_planar[4]()

HEVCPredContext -> pred_planar[4]()指向了帧内预测Planar模式的汇编函数。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体C语言版本处理函数为：

pred_planar_0_8()——4x4块；

pred_planar_1_8()——8x8块；

pred_planar_2_8()——16x16块；

pred_planar_3_8()——32x32块；

这四个函数的定义如下所示。

```cpp
#define PRED_PLANAR(size)\
static void FUNC(pred_planar_ ## size)(uint8_t *src, const uint8_t *top,        \
                                       const uint8_t *left, ptrdiff_t stride)   \
{                                                                               \
    FUNC(pred_planar)(src, top, left, stride, size + 2);                        \
}
/* 几种不同大小的方块对应的Planar预测函数
 * 参数取值越大，代表的方块越大：
 * [0]:4x4块
 * [1]:8x8块
 * [2]:16x16块
 * [3]:32x32块
 *
 * “PRED_PLANAR(0)”展开后的函数是
 * static void pred_planar_0_8(uint8_t *src, const uint8_t *top,
 *                                       const uint8_t *left, ptrdiff_t stride)
 * {
 *     pred_planar_8(src, top, left, stride, 0 + 2);
 * }
 */
PRED_PLANAR(0)
PRED_PLANAR(1)
PRED_PLANAR(2)
PRED_PLANAR(3)
```

从源代码中可以看出，pred_planar_0_8()、pred_planar_1_8()等函数都是通过“PRED_PLANAR ()”宏进行定义的。pred_planar_0_8()、pred_planar_1_8()等函数的内部都调用了同一个函数pred_planar_8()。这几个函数唯一的不同在于，调用intra_pred_8()时候第5个参数trafo_size的值不一样。

pred_planar_8()

pred_planar_8()实现了Planar帧内预测模式，该函数的定义如下所示。

```cpp
#define POS(x, y) src[(x) + stride * (y)]

//Planar预测模式
static av_always_inline void FUNC(pred_planar)(uint8_t *_src, const uint8_t *_top,
                                  const uint8_t *_left, ptrdiff_t stride,
                                  int trafo_size)
{
    int x, y;
    pixel *src        = (pixel *)_src;
    //上面1行像素
    const pixel *top  = (const pixel *)_top;
    //左边1列像素
    const pixel *left = (const pixel *)_left;
    int size = 1 << trafo_size;
    //双线性插值
    //注意[size]为最后一个元素
    for (y = 0; y < size; y++)
        for (x = 0; x < size; x++)
            POS(x, y) = ((size - 1 - x) * left[y] + (x + 1) * top[size]  +
                         (size - 1 - y) * top[x]  + (y + 1) * left[size] + size) >> (trafo_size + 1);
}
```

从源代码可以看出，pred_planar_8()以一种类似双线性插值的方式完成了预测数据的填充。其中src指向方块的像素区域，left指向方块左边一列像素，top指向方块上边一行像素。Planar模式的计算方式如下图所示。

从图中可以看出，Planar模式首先将左边一列像素最下面一个像素值水平复制一行，将上边一行像素最右边一个像素值垂直复制一列；然后使用类似于双线性插值的方式，获得预测数据。

HEVCPredContext -> pred_dc ()

HEVCPredContext -> pred_dc()指向了帧内预测DC模式的汇编函数。具体的C语言版本的处理函数是pred_dc_8()。pred_dc_8()的定义如下。

[cpp]

```cpp
#define POS(x, y) src[(x) + stride * (y)]

//DC预测模式
static void FUNC(pred_dc)(uint8_t *_src, const uint8_t *_top,
                          const uint8_t *_left,
                          ptrdiff_t stride, int log2_size, int c_idx)
{
    int i, j, x, y;
    int size          = (1 << log2_size);
    pixel *src        = (pixel *)_src;
    const pixel *top  = (const pixel *)_top;
    const pixel *left = (const pixel *)_left;
    int dc            = size;
    //pixel4为unit32_t，即存储了4个像素
    pixel4 a;
    //累加左边1列，和上边1行
    for (i = 0; i < size; i++)
        dc += left[i] + top[i];
    //求平均
    dc >>= log2_size + 1;
    //取出来值
    a = PIXEL_SPLAT_X4(dc);
    //赋值到像素块中的每个点
    for (i = 0; i < size; i++)
        for (j = 0; j < size; j+=4)
            AV_WN4P(&POS(j, i), a);

    if (c_idx == 0 && size < 32) {
        POS(0, 0) = (left[0] + 2 * dc + top[0] + 2) >> 2;
        for (x = 1; x < size; x++)
            POS(x, 0) = (top[x] + 3 * dc + 2) >> 2;
        for (y = 1; y < size; y++)
            POS(0, y) = (left[y] + 3 * dc + 2) >> 2;
    }
}
```

从源代码可以看出，pred_dc_8()首先求得了左边一行像素和上边一行像素的平均值，然后将该值作为预测数据赋值给了整个方块。

HEVCPredContext -> pred_angular ()

HEVCPredContext -> pred_angular[4]()指向了帧内预测角度（Angular）模式的汇编函数。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体C语言版本处理函数为：

pred_angular_0_8()——4x4块；

pred_angular_1_8()——8x8块；

pred_angular_2_8()——16x16块；

pred_angular_3_8()——32x32块；

这四个函数的定义如下所示。

```cpp
/* 几种不同大小的方块对应的Angular预测函数
 * 数字取值越大，代表的方块越大：
 * [0]:4x4块
 * [1]:8x8块
 * [2]:16x16块
 * [3]:32x32块
 *
 */
static void FUNC(pred_angular_0)(uint8_t *src, const uint8_t *top,
                                 const uint8_t *left,
                                 ptrdiff_t stride, int c_idx, int mode)
{
    FUNC(pred_angular)(src, top, left, stride, c_idx, mode, 1 << 2);
}

static void FUNC(pred_angular_1)(uint8_t *src, const uint8_t *top,
                                 const uint8_t *left,
                                 ptrdiff_t stride, int c_idx, int mode)
{
    FUNC(pred_angular)(src, top, left, stride, c_idx, mode, 1 << 3);
}

static void FUNC(pred_angular_2)(uint8_t *src, const uint8_t *top,
                                 const uint8_t *left,
                                 ptrdiff_t stride, int c_idx, int mode)
{
    FUNC(pred_angular)(src, top, left, stride, c_idx, mode, 1 << 4);
}

static void FUNC(pred_angular_3)(uint8_t *src, const uint8_t *top,
                                 const uint8_t *left,
                                 ptrdiff_t stride, int c_idx, int mode)
{
    FUNC(pred_angular)(src, top, left, stride, c_idx, mode, 1 << 5);
}
```

从源代码可以看出，pred_angular_0_8()、pred_angular_1_8()等函数的内部都调用了同样的一个函数pred_angular_8()。它们之间的不同在于传递给pred_angular_8()的最后一个参数size取值的不同。

pred_angular_8()

pred_planar_8()实现了角度（Angular）帧内预测模式，该函数的定义如下所示。

```cpp
#define POS(x, y) src[(x) + stride * (y)]

static av_always_inline void FUNC(pred_angular)(uint8_t *_src,
                                                const uint8_t *_top,
                                                const uint8_t *_left,
                                                ptrdiff_t stride, int c_idx,
                                                int mode, int size)
{
    int x, y;
    pixel *src        = (pixel *)_src;
    const pixel *top  = (const pixel *)_top;
    const pixel *left = (const pixel *)_left;
    //角度
    static const int intra_pred_angle[] = {
         32,  26,  21,  17, 13,  9,  5, 2, 0, -2, -5, -9, -13, -17, -21, -26, -32,
        -26, -21, -17, -13, -9, -5, -2, 0, 2,  5,  9, 13,  17,  21,  26,  32
    };
    static const int inv_angle[] = {
        -4096, -1638, -910, -630, -482, -390, -315, -256, -315, -390, -482,
        -630, -910, -1638, -4096
    };
    //mode的前两种是Planar和DC，不属于角度预测
    int angle = intra_pred_angle[mode - 2];
    pixel ref_array[3 * MAX_TB_SIZE + 4];
    pixel *ref_tmp = ref_array + size;
    const pixel *ref;
    int last = (size * angle) >> 5;

    if (mode >= 18) {
        //垂直类模式

        ref = top - 1;
        if (angle < 0 && last < -1) {
            for (x = 0; x <= size; x += 4)
                AV_WN4P(&ref_tmp[x], AV_RN4P(&top[x - 1]));
            for (x = last; x <= -1; x++)
                ref_tmp[x] = left[-1 + ((x * inv_angle[mode - 11] + 128) >> 8)];
            ref = ref_tmp;
        }

        for (y = 0; y < size; y++) {
            int idx  = ((y + 1) * angle) >> 5;
            int fact = ((y + 1) * angle) & 31;
            if (fact) {
```

```cpp
            if (fact) {
                for (x = 0; x < size; x += 4) {
                    POS(x    , y) = ((32 - fact) * ref[x + idx + 1] +
                                           fact  * ref[x + idx + 2] + 16) >> 5;
                    POS(x + 1, y) = ((32 - fact) * ref[x + 1 + idx + 1] +
                                           fact  * ref[x + 1 + idx + 2] + 16) >> 5;
                    POS(x + 2, y) = ((32 - fact) * ref[x + 2 + idx + 1] +
                                           fact  * ref[x + 2 + idx + 2] + 16) >> 5;
                    POS(x + 3, y) = ((32 - fact) * ref[x + 3 + idx + 1] +
                                           fact  * ref[x + 3 + idx + 2] + 16) >> 5;
                }
            } else {
                for (x = 0; x < size; x += 4)
                    AV_WN4P(&POS(x, y), AV_RN4P(&ref[x + idx + 1]));
            }
        }
        if (mode == 26 && c_idx == 0 && size < 32) {
            for (y = 0; y < size; y++)
                POS(0, y) = av_clip_pixel(top[0] + ((left[y] - left[-1]) >> 1));
        }
    } else {
        //水平类模式

        ref = left - 1;
        if (angle < 0 && last < -1) {
            for (x = 0; x <= size; x += 4)
                AV_WN4P(&ref_tmp[x], AV_RN4P(&left[x - 1]));
            for (x = last; x <= -1; x++)
                ref_tmp[x] = top[-1 + ((x * inv_angle[mode - 11] + 128) >> 8)];
            ref = ref_tmp;
        }

        for (x = 0; x < size; x++) {
            int idx  = ((x + 1) * angle) >> 5;
            int fact = ((x + 1) * angle) & 31;
            if (fact) {
                for (y = 0; y < size; y++) {
                    POS(x, y) = ((32 - fact) * ref[y + idx + 1] +
                                       fact  * ref[y + idx + 2] + 16) >> 5;
                }
            } else {
                for (y = 0; y < size; y++)
                    POS(x, y) = ref[y + idx + 1];
            }
        }
        if (mode == 10 && c_idx == 0 && size < 32) {
            for (x = 0; x < size; x += 4) {
                POS(x,     0) = av_clip_pixel(left[0] + ((top[x    ] - top[-1]) >> 1));
                POS(x + 1, 0) = av_clip_pixel(left[0] + ((top[x + 1] - top[-1]) >> 1));
                POS(x + 2, 0) = av_clip_pixel(left[0] + ((top[x + 2] - top[-1]) >> 1));
                POS(x + 3, 0) = av_clip_pixel(left[0] + ((top[x + 3] - top[-1]) >> 1));
            }
        }
    }
}
```

pred_planar_8()的代码还没有细看，以后有时间再做分析。
至此有关帧内预测方面的源代码就基本分析完了。后文继续分析DCT反变换相关的源代码。

DCT反变换汇编函数源代码

DCT反变换相关的汇编函数位于HEVCDSPContext中。HEVCDSPContext的初始化函数是ff_hevc_dsp_init()。该函数对HEVCDSPContext结构体中的函数指针进行了赋值。FFmpeg HEVC解码器运行的过程中只要调用HEVCDSPContext的函数指针就可以完成相应的功能。

ff_hevc_dsp_init()

ff_hevc_dsp_init()用于初始化HEVCDSPContext结构体中的汇编函数指针。该函数的定义如下所示。

```cpp
void ff_hevc_dsp_init(HEVCDSPContext *hevcdsp, int bit_depth)
{
#undef FUNC
#define FUNC(a, depth) a ## _ ## depth

#undef PEL_FUNC
#define PEL_FUNC(dst1, idx1, idx2, a, depth)                                   \
    for(i = 0 ; i < 10 ; i++)                                                  \
{                                                                              \
    hevcdsp->dst1[i][idx1][idx2] = a ## _ ## depth;                            \
}

#undef EPEL_FUNCS
#define EPEL_FUNCS(depth)                                                     \
    PEL_FUNC(put_hevc_epel, 0, 0, put_hevc_pel_pixels, depth);                \
    PEL_FUNC(put_hevc_epel, 0, 1, put_hevc_epel_h, depth);                    \
```

```
    PEL_FUNC(put_hevc_epel, 0, 1, put_hevc_epel_h, depth);                 \
    PEL_FUNC(put_hevc_epel, 1, 0, put_hevc_epel_v, depth);                 \
    PEL_FUNC(put_hevc_epel, 1, 1, put_hevc_epel_hv, depth)

#undef EPEL_UNI_FUNCS
#define EPEL_UNI_FUNCS(depth)                                              \
    PEL_FUNC(put_hevc_epel_uni, 0, 0, put_hevc_pel_uni_pixels, depth);     \
    PEL_FUNC(put_hevc_epel_uni, 0, 1, put_hevc_epel_uni_h, depth);         \
    PEL_FUNC(put_hevc_epel_uni, 1, 0, put_hevc_epel_uni_v, depth);         \
    PEL_FUNC(put_hevc_epel_uni, 1, 1, put_hevc_epel_uni_hv, depth);        \
    PEL_FUNC(put_hevc_epel_uni_w, 0, 0, put_hevc_pel_uni_w_pixels, depth); \
    PEL_FUNC(put_hevc_epel_uni_w, 0, 1, put_hevc_epel_uni_w_h, depth);     \
    PEL_FUNC(put_hevc_epel_uni_w, 1, 0, put_hevc_epel_uni_w_v, depth);     \
    PEL_FUNC(put_hevc_epel_uni_w, 1, 1, put_hevc_epel_uni_w_hv, depth)

#undef EPEL_BI_FUNCS
#define EPEL_BI_FUNCS(depth)                                             \
    PEL_FUNC(put_hevc_epel_bi, 0, 0, put_hevc_pel_bi_pixels, depth);     \
    PEL_FUNC(put_hevc_epel_bi, 0, 1, put_hevc_epel_bi_h, depth);         \
    PEL_FUNC(put_hevc_epel_bi, 1, 0, put_hevc_epel_bi_v, depth);         \
    PEL_FUNC(put_hevc_epel_bi, 1, 1, put_hevc_epel_bi_hv, depth);        \
    PEL_FUNC(put_hevc_epel_bi_w, 0, 0, put_hevc_pel_bi_w_pixels, depth); \
    PEL_FUNC(put_hevc_epel_bi_w, 0, 1, put_hevc_epel_bi_w_h, depth);     \
    PEL_FUNC(put_hevc_epel_bi_w, 1, 0, put_hevc_epel_bi_w_v, depth);     \
    PEL_FUNC(put_hevc_epel_bi_w, 1, 1, put_hevc_epel_bi_w_hv, depth)

#undef QPEL_FUNCS
#define QPEL_FUNCS(depth)                                                  \
    PEL_FUNC(put_hevc_qpel, 0, 0, put_hevc_pel_pixels, depth);             \
    PEL_FUNC(put_hevc_qpel, 0, 1, put_hevc_qpel_h, depth);                 \
    PEL_FUNC(put_hevc_qpel, 1, 0, put_hevc_qpel_v, depth);                 \
    PEL_FUNC(put_hevc_qpel, 1, 1, put_hevc_qpel_hv, depth)

#undef QPEL_UNI_FUNCS
#define QPEL_UNI_FUNCS(depth)                                              \
    PEL_FUNC(put_hevc_qpel_uni, 0, 0, put_hevc_pel_uni_pixels, depth);     \
    PEL_FUNC(put_hevc_qpel_uni, 0, 1, put_hevc_qpel_uni_h, depth);         \
    PEL_FUNC(put_hevc_qpel_uni, 1, 0, put_hevc_qpel_uni_v, depth);         \
    PEL_FUNC(put_hevc_qpel_uni, 1, 1, put_hevc_qpel_uni_hv, depth);        \
    PEL_FUNC(put_hevc_qpel_uni_w, 0, 0, put_hevc_pel_uni_w_pixels, depth); \
    PEL_FUNC(put_hevc_qpel_uni_w, 0, 1, put_hevc_qpel_uni_w_h, depth);     \
    PEL_FUNC(put_hevc_qpel_uni_w, 1, 0, put_hevc_qpel_uni_w_v, depth);     \
    PEL_FUNC(put_hevc_qpel_uni_w, 1, 1, put_hevc_qpel_uni_w_hv, depth)

#undef QPEL_BI_FUNCS
#define QPEL_BI_FUNCS(depth)                                               \
    PEL_FUNC(put_hevc_qpel_bi, 0, 0, put_hevc_pel_bi_pixels, depth);       \
    PEL_FUNC(put_hevc_qpel_bi, 0, 1, put_hevc_qpel_bi_h, depth);           \
    PEL_FUNC(put_hevc_qpel_bi, 1, 0, put_hevc_qpel_bi_v, depth);           \
    PEL_FUNC(put_hevc_qpel_bi, 1, 1, put_hevc_qpel_bi_hv, depth);          \
    PEL_FUNC(put_hevc_qpel_bi_w, 0, 0, put_hevc_pel_bi_w_pixels, depth);   \
    PEL_FUNC(put_hevc_qpel_bi_w, 0, 1, put_hevc_qpel_bi_w_h, depth);       \
    PEL_FUNC(put_hevc_qpel_bi_w, 1, 0, put_hevc_qpel_bi_w_v, depth);       \
    PEL_FUNC(put_hevc_qpel_bi_w, 1, 1, put_hevc_qpel_bi_w_hv, depth)

#define HEVC_DSP(depth)                                                  \
    hevcdsp->put_pcm                = FUNC(put_pcm, depth);              \
    hevcdsp->transform_add[0]       = FUNC(transform_add4x4, depth);     \
    hevcdsp->transform_add[1]       = FUNC(transform_add8x8, depth);     \
    hevcdsp->transform_add[2]       = FUNC(transform_add16x16, depth);   \
    hevcdsp->transform_add[3]       = FUNC(transform_add32x32, depth);   \
    hevcdsp->transform_skip         = FUNC(transform_skip, depth);       \
    hevcdsp->transform_rdpcm        = FUNC(transform_rdpcm, depth);      \
    hevcdsp->idct_4x4_luma          = FUNC(transform_4x4_luma, depth);   \
    hevcdsp->idct[0]                = FUNC(idct_4x4, depth);             \
    hevcdsp->idct[1]                = FUNC(idct_8x8, depth);             \
    hevcdsp->idct[2]                = FUNC(idct_16x16, depth);           \
    hevcdsp->idct[3]                = FUNC(idct_32x32, depth);           \
                                                                         \
    hevcdsp->idct_dc[0]             = FUNC(idct_4x4_dc, depth);          \
    hevcdsp->idct_dc[1]             = FUNC(idct_8x8_dc, depth);          \
    hevcdsp->idct_dc[2]             = FUNC(idct_16x16_dc, depth);        \
    hevcdsp->idct_dc[3]             = FUNC(idct_32x32_dc, depth);        \
                                                                         \
    hevcdsp->sao_band_filter    = FUNC(sao_band_filter_0, depth);           \
    hevcdsp->sao_edge_filter[0] = FUNC(sao_edge_filter_0, depth);           \
    hevcdsp->sao_edge_filter[1] = FUNC(sao_edge_filter_1, depth);           \
                                                                            \
    QPEL_FUNCS(depth);                                                      \
    QPEL_UNI_FUNCS(depth);                                                  \
    QPEL_BI_FUNCS(depth);                                                   \
    EPEL_FUNCS(depth);                                                      \
    EPEL_UNI_FUNCS(depth);                                                  \
    EPEL_BI_FUNCS(depth);                                                   \
                                                                            \
    hevcdsp->hevc_h_loop_filter_luma     = FUNC(hevc_h_loop_filter_luma, depth);   \
    hevcdsp->hevc_v_loop_filter_luma     = FUNC(hevc_v_loop_filter_luma, depth);   \
    hevcdsp->hevc_h_loop_filter_chroma   = FUNC(hevc_h_loop_filter_chroma, depth); \
    hevcdsp->hevc_v_loop_filter_chroma   = FUNC(hevc_v_loop_filter_chroma, depth); \
    hevcdsp->hevc_h_loop_filter_luma_c   = FUNC(hevc_h_loop_filter_luma, depth);   \
    hevcdsp->hevc_v_loop_filter_luma_c   = FUNC(hevc_v_loop_filter_luma, depth);   \
    hevcdsp->hevc_h_loop_filter_chroma_c = FUNC(hevc_h_loop_filter_chroma, depth); \
```

```
    hevcdsp->hevc_v_loop_filter_chroma_c = FUNC(hevc_v_loop_filter_chroma, depth)
int i = 0;

    switch (bit_depth) {
    case 9:
        HEVC_DSP(9);
        break;
    case 10:
        HEVC_DSP(10);
        break;
    case 12:
        HEVC_DSP(12);
        break;
    default:
        HEVC_DSP(8);
        break;
    }

    if (ARCH_X86)
        ff_hevc_dsp_init_x86(hevcdsp, bit_depth);
}
```

从源代码可以看出，ff_hevc_dsp_init()函数中包含一个名为“HEVC_DSP(depth)”的很长的宏定义。该宏定义中包含了C语言版本的各种函数的初始化代码。ff_hevc_dsp_init()会根据系统的颜色位深bit_depth初始化相应的C语言版本的函数。在函数的末尾则包含了汇编函数的初始化函数：如果系统是X86架构的，则会调用ff_hevc_dsp_init_x86()初始化X86平台下经过汇编优化的函数。下面以8bit颜色位深为例，看一下“HEVC_DSP(8)”的展开结果中和DCT相关的函数。

```cpp
hevcdsp->transform_add[0]        = transform_add4x4_8;
hevcdsp->transform_add[1]        = transform_add8x8_8;
hevcdsp->transform_add[2]        = transform_add16x16_8;
hevcdsp->transform_add[3]        = transform_add32x32_8;
hevcdsp->transform_skip          = transform_skip_8;
hevcdsp->transform_rdpcm         = transform_rdpcm_8;
hevcdsp->idct_4x4_luma           = transform_4x4_luma_8;
hevcdsp->idct[0]                 = idct_4x4_8;
hevcdsp->idct[1]                 = idct_8x8_8;
hevcdsp->idct[2]                 = idct_16x16_8;
hevcdsp->idct[3]                 = idct_32x32_8;

hevcdsp->idct_dc[0]              = idct_4x4_dc_8;
hevcdsp->idct_dc[1]              = idct_8x8_dc_8;
hevcdsp->idct_dc[2]              = idct_16x16_dc_8;
hevcdsp->idct_dc[3]              = idct_32x32_dc_8;
//略….                     <span style="font-family: Arial, Helvetica, sans-serif;">                                     </span>
```

通过上述代码可以总结出下面几个IDCT函数（数组）：

HEVCDSPContext -> idct[4]()：DCT反变换函数。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

HEVCDSPContext -> idct_dc[4]() ：只有DC系数时候的DCT反变换函数（运算速度比普通DCT快一些）。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

HEVCDSPContext -> idct_4x4_luma()：特殊的4x4DST反变换函数。在处理Intra4x4块的时候，HEVC使用了一种比较特殊的DST（而不是DCT），可以微量的提高编码效率。

HEVCDSPContext -> transform_add[4]()：残差叠加函数，用于将IDCT之后的残差像素数据叠加到预测像素数据之上。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

PS：还有几种IDCT函数目前还没有看，就不列出了。

下面分别看一下上述的几种函数。

HEVCDSPContext -> idct[4]()

HEVCPredContext -> idct[4]()指向了DCT反变换的汇编函数。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体C语言版本处理函数为：

idct_4x4_8()——4x4块；

idct_8x8_8()——8x8块；

idct_16x16_8()——16x16块；

idct_32x32_8()——32x32块；

这四个函数的定义如下所示。

```cpp
#define SET(dst, x)   (dst) = (x)
#define SCALE(dst, x) (dst) = av_clip_int16(((x) + add) >> shift)
#define ADD_AND_SCALE(dst, x)                                    \
    (dst) = av_clip_pixel((dst) + av_clip_int16(((x) + add) >> shift))

#define IDCT_VAR4(H)                                                 \
    int      limit2   = FFMIN(col_limit + 4, H)
#define IDCT_VAR8(H)                                                 \
        int      limit    = FFMIN(col_limit, H);                     \
        int      limit2   = FFMIN(col_limit + 4, H)
#define IDCT_VAR16(H)   IDCT_VAR8(H)
#define IDCT_VAR32(H)   IDCT_VAR8(H)

//其中的“H”取4,8,16,32
//可以拼凑出不同的函数
#define IDCT(H)                                                     \
static void FUNC(idct_##H ##x ##H )(                                \
                  int16_t *coeffs, int col_limit) {                 \
    int i;                                                          \
    int      shift   = 7;                                           \
    int      add     = 1 << (shift - 1);                            \
    int16_t *src     = coeffs;                                      \
    IDCT_VAR ##H(H);                                                \
                                                                    \
    for (i = 0; i < H; i++) {                                       \
        TR_ ## H(src, src, H, H, SCALE, limit2);                    \
        if (limit2 < H && i%4 == 0 && !!i)                          \
            limit2 -= 4;                                            \
        src++;                                                      \
    }                                                               \
                                                                    \
    shift   = 20 - BIT_DEPTH;                                       \
    add     = 1 << (shift - 1);                                     \
    for (i = 0; i < H; i++) {                                       \
        TR_ ## H(coeffs, coeffs, 1, 1, SCALE, limit);               \
        coeffs += H;                                                \
    }                                                               \
}

//几种不同尺度的IDCT
IDCT( 4)
IDCT( 8)
IDCT(16)
IDCT(32)
```

从源代码可以看出，idct_4x4_8()、idct_8x8_8()等函数的定义是通过“IDCT()”宏实现的。而“IDCT(H)”宏中又调用了另外一个宏“TR_ ## H()”。“TR_ ## H()”根据“H”取值的不同，可以调用：

TR_4()——用于4x4DCT

TR_8()——用于8x8DCT

TR_16()——用于16x16DCT

TR_32()——用于32x32DCT

TR4()、TR8()、TR16()、TR32()的定义如下所示。

[cpp]

```cpp
/*
 * 4x4DCT
 *
 *      | 64  64  64  64 |
 * H =  | 83  36 -36 -83 |
 *      | 64 -64 -64  64 |
 *      | 36 -83  83 -36 |
 *
 */
#define TR_4(dst, src, dstep, sstep, assign, end)                       \
    do {                                                                \
        const int e0 = 64 * src[0 * sstep] + 64 * src[2 * sstep];       \
        const int e1 = 64 * src[0 * sstep] - 64 * src[2 * sstep];       \
        const int o0 = 83 * src[1 * sstep] + 36 * src[3 * sstep];       \
        const int o1 = 36 * src[1 * sstep] - 83 * src[3 * sstep];       \
                                                                        \
        assign(dst[0 * dstep], e0 + o0);                                \
        assign(dst[1 * dstep], e1 + o1);                                \
        assign(dst[2 * dstep], e1 - o1);                                \
        assign(dst[3 * dstep], e0 - o0);                                \
    } while (0)

/*
 * 8x8DCT
 *
 * transform[]存储了32x32DCT变换系数
 * 8x8DCT变换的系数来自于32x32系数矩阵中第0，4，8，12，16，20，24，28行元素中的前8个元素
 *
 */
#define TR_8(dst, src, dstep, sstep, assign, end)                       \
    do {                                                                \
        int i, j;                                                       \
        int e_8[4];                                                     \
        int o_8[4] = { 0 };                                             \
        for (i = 0; i < 4; i++)                                         \
            for (j = 1; j < end; j += 2)                                \
                o_8[i] += transform[4 * j][i] * src[j * sstep];         \
        TR_4(e_8, src, 1, 2 * sstep, SET, 4);                           \
                                                                        \
        for (i = 0; i < 4; i++) {                                       \
            assign(dst[i * dstep], e_8[i] + o_8[i]);                    \
            assign(dst[(7 - i) * dstep], e_8[i] - o_8[i]);              \
        }                                                               \
    } while (0)

/*
 * 16x16DCT
 * 16x16 DCT变换的系数来自于32x32系数矩阵中第0，2，4…，28，30行元素中的前16个元素
 *
 */
#define TR_16(dst, src, dstep, sstep, assign, end)                      \
    do {                                                                \
        int i, j;                                                       \
        int e_16[8];                                                    \
        int o_16[8] = { 0 };                                            \
        for (i = 0; i < 8; i++)                                         \
            for (j = 1; j < end; j += 2)                                \
                o_16[i] += transform[2 * j][i] * src[j * sstep];        \
        TR_8(e_16, src, 1, 2 * sstep, SET, 8);                          \
                                                                        \
        for (i = 0; i < 8; i++) {                                       \
            assign(dst[i * dstep], e_16[i] + o_16[i]);                  \
            assign(dst[(15 - i) * dstep], e_16[i] - o_16[i]);           \
        }                                                               \
    } while (0)

/*
 * 32x32DCT
 *
 */
#define TR_32(dst, src, dstep, sstep, assign, end)                      \
    do {                                                                \
        int i, j;                                                       \
        int e_32[16];                                                   \
        int o_32[16] = { 0 };                                           \
        for (i = 0; i < 16; i++)                                        \
            for (j = 1; j < end; j += 2)                                \
                o_32[i] += transform[j][i] * src[j * sstep];            \
        TR_16(e_32, src, 1, 2 * sstep, SET, end/2);                     \
                                                                        \
        for (i = 0; i < 16; i++) {                                      \
            assign(dst[i * dstep], e_32[i] + o_32[i]);                  \
            assign(dst[(31 - i) * dstep], e_32[i] - o_32[i]);           \
        }                                                               \
    } while (0)
```

有关这一部分的源代码目前还没有细看，以后有时间再进行补充。从TR8()、TR16()等的定义中可以看出，它们的DCT系数来自于一个transform[32][32]数组。

transform[32][32]

transform[32][32] 的定义如下所示，其中存储了32x32DCT的系数。使用该系数矩阵，也可以推导获得16x16DCT、8x8DCT、4x4DCT的系数。

```cpp
//32x32DCT变换系数
static const int8_t transform[32][32] = {
    { 64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,
      64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64 },
    { 90,  90,  88,  85,  82,  78,  73,  67,  61,  54,  46,  38,  31,  22,  13,   4,
      -4, -13, -22, -31, -38, -46, -54, -61, -67, -73, -78, -82, -85, -88, -90, -90 },
    { 90,  87,  80,  70,  57,  43,  25,   9,  -9, -25, -43, -57, -70, -80, -87, -90,
     -90, -87, -80, -70, -57, -43, -25,  -9,   9,  25,  43,  57,  70,  80,  87,  90 },
    { 90,  82,  67,  46,  22,  -4, -31, -54, -73, -85, -90, -88, -78, -61, -38, -13,
      13,  38,  61,  78,  88,  90,  85,  73,  54,  31,   4, -22, -46, -67, -82, -90 },
    { 89,  75,  50,  18, -18, -50, -75, -89, -89, -75, -50, -18,  18,  50,  75,  89,
      89,  75,  50,  18, -18, -50, -75, -89, -89, -75, -50, -18,  18,  50,  75,  89 },
    { 88,  67,  31, -13, -54, -82, -90, -78, -46, -4,   38,  73,  90,  85,  61,  22,
     -22, -61, -85, -90, -73, -38,   4,  46,  78,  90,  82,  54,  13, -31, -67, -88 },
    { 87,  57,   9, -43, -80, -90, -70, -25,  25,  70,  90,  80,  43,  -9, -57, -87,
     -87, -57,  -9,  43,  80,  90,  70,  25, -25, -70, -90, -80, -43,   9,  57,  87 },
    { 85,  46, -13, -67, -90, -73, -22,  38,  82,  88,  54,  -4, -61, -90, -78, -31,
      31,  78,  90,  61,   4, -54, -88, -82, -38,  22,  73,  90,  67,  13, -46, -85 },
    { 83,  36, -36, -83, -83, -36,  36,  83,  83,  36, -36, -83, -83, -36,  36,  83,
      83,  36, -36, -83, -83, -36,  36,  83,  83,  36, -36, -83, -83, -36,  36,  83 },
    { 82,  22, -54, -90, -61,  13,  78,  85,  31, -46, -90, -67,   4,  73,  88,  38,
     -38, -88, -73,  -4,  67,  90,  46, -31, -85, -78, -13,  61,  90,  54, -22, -82 },
    { 80,   9, -70, -87, -25,  57,  90,  43, -43, -90, -57,  25,  87,  70,  -9, -80,
     -80,  -9,  70,  87,  25, -57, -90, -43,  43,  90,  57, -25, -87, -70,   9,  80 },
    { 78,  -4, -82, -73,  13,  85,  67, -22, -88, -61,  31,  90,  54, -38, -90, -46,
      46,  90,  38, -54, -90, -31,  61,  88,  22, -67, -85, -13,  73,  82,   4, -78 },
    { 75, -18, -89, -50,  50,  89,  18, -75, -75,  18,  89,  50, -50, -89, -18,  75,
      75, -18, -89, -50,  50,  89,  18, -75, -75,  18,  89,  50, -50, -89, -18,  75 },
    { 73, -31, -90, -22,  78,  67, -38, -90, -13,  82,  61, -46, -88,  -4,  85,  54,
     -54, -85,   4,  88,  46, -61, -82,  13,  90,  38, -67, -78,  22,  90,  31, -73 },
    { 70, -43, -87,   9,  90,  25, -80, -57,  57,  80, -25, -90,  -9,  87,  43, -70,
     -70,  43,  87,  -9, -90, -25,  80,  57, -57, -80,  25,  90,   9, -87, -43,  70 },
    { 67, -54, -78,  38,  85, -22, -90,   4,  90,  13, -88, -31,  82,  46, -73, -61,
      61,  73, -46, -82,  31,  88, -13, -90,  -4,  90,  22, -85, -38,  78,  54, -67 },
    { 64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64,
      64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64 },
    { 61, -73, -46,  82,  31, -88, -13,  90,  -4, -90,  22,  85, -38, -78,  54,  67,
     -67, -54,  78,  38, -85, -22,  90,   4, -90,  13,  88, -31, -82,  46,  73, -61 },
    { 57, -80, -25,  90,  -9, -87,  43,  70, -70, -43,  87,   9, -90,  25,  80, -57,
     -57,  80,  25, -90,   9,  87, -43, -70,  70,  43, -87,  -9,  90, -25, -80,  57 },
    { 54, -85,  -4,  88, -46, -61,  82,  13, -90,  38,  67, -78, -22,  90, -31, -73,
      73,  31, -90,  22,  78, -67, -38,  90, -13, -82,  61,  46, -88,   4,  85, -54 },
    { 50, -89,  18,  75, -75, -18,  89, -50, -50,  89, -18, -75,  75,  18, -89,  50,
      50, -89,  18,  75, -75, -18,  89, -50, -50,  89, -18, -75,  75,  18, -89,  50 },
    { 46, -90,  38,  54, -90,  31,  61, -88,  22,  67, -85,  13,  73, -82,   4,  78,
     -78,  -4,  82, -73, -13,  85, -67, -22,  88, -61, -31,  90, -54, -38,  90, -46 },
    { 43, -90,  57,  25, -87,  70,   9, -80,  80,  -9, -70,  87, -25, -57,  90, -43,
     -43,  90, -57, -25,  87, -70,  -9,  80, -80,   9,  70, -87,  25,  57, -90,  43 },
    { 38, -88,  73,  -4, -67,  90, -46, -31,  85, -78,  13,  61, -90,  54,  22, -82,
      82, -22, -54,  90, -61, -13,  78, -85,  31,  46, -90,  67,   4, -73,  88, -38 },
    { 36, -83,  83, -36, -36,  83, -83,  36,  36, -83,  83, -36, -36,  83, -83,  36,
      36, -83,  83, -36, -36,  83, -83,  36,  36, -83,  83, -36, -36,  83, -83,  36 },
    { 31, -78,  90, -61,   4,  54, -88,  82, -38, -22,  73, -90,  67, -13, -46,  85,
     -85,  46,  13, -67,  90, -73,  22,  38, -82,  88, -54,  -4,  61, -90,  78, -31 },
    { 25, -70,  90, -80,  43,   9, -57,  87, -87,  57,  -9, -43,  80, -90,  70, -25,
     -25,  70, -90,  80, -43,  -9,  57, -87,  87, -57,   9,  43, -80,  90, -70,  25 },
    { 22, -61,  85, -90,  73, -38,  -4,  46, -78,  90, -82,  54, -13, -31,  67, -88,
      88, -67,  31,  13, -54,  82, -90,  78, -46,   4,  38, -73,  90, -85,  61, -22 },
    { 18, -50,  75, -89,  89, -75,  50, -18, -18,  50, -75,  89, -89,  75, -50,  18,
      18, -50,  75, -89,  89, -75,  50, -18, -18,  50, -75,  89, -89,  75, -50,  18 },
    { 13, -38,  61, -78,  88, -90,  85, -73,  54, -31,   4,  22, -46,  67, -82,  90,
     -90,  82, -67,  46, -22,  -4,  31, -54,  73, -85,  90, -88,  78, -61,  38, -13 },
    {  9, -25,  43, -57,  70, -80,  87, -90,  90, -87,  80, -70,  57, -43,  25, -9,
      -9,  25, -43,  57, -70,  80, -87,  90, -90,  87, -80,  70, -57,  43, -25,   9 },
    {  4, -13,  22, -31,  38, -46,  54, -61,  67, -73,  78, -82,  85, -88,  90, -90,
      90, -90,  88, -85,  82, -78,  73, -67,  61, -54,  46, -38,  31, -22,  13,  -4 },
};
```

HEVCDSPContext -> idct_dc[4]()

HEVCPredContext -> idct_dc[4]()指向了只有DC系数时候的DCT反变换的汇编函数。只有DC系数的DCT反变换属于一种比较特殊的情况，在这种情况下使用idct_dc[4]()的速度会比idct[4]()要快一些。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体C语言版本处理函数为：

idct_4x4_dc_8()——4x4块；

idct_8x8_dc_8()——8x8块；

idct_16x16_dc_8()——16x16块；

idct_32x32_dc_8()——32x32块；

这四个函数的定义如下所示。

```cpp
#define IDCT_DC(H)                                                    \
static void FUNC(idct_##H ##x ##H ##_dc)(                             \
                 int16_t *coeffs) {                                   \
    int i, j;                                                         \
    int      shift   = 14 - BIT_DEPTH;                                \
    int      add     = 1 << (shift - 1);                              \
    int      coeff   = (((coeffs[0] + 1) >> 1) + add) >> shift;       \
                                                                      \
    for (j = 0; j < H; j++) {                                         \
        for (i = 0; i < H; i++) {                                     \
            coeffs[i+j*H] = coeff;                                    \
        }                                                             \
    }                                                                 \
}


//只包含DC系数时候的比较快速的IDCT
IDCT_DC( 4)
IDCT_DC( 8)
IDCT_DC(16)
IDCT_DC(32)
```

可以看出idct_4x4_dc_8()、idct_8x8_dc_8()等函数的初始化是通过“IDCT_DC()”宏完成的。可以看出“IDCT_DC()”首先通过DC系数coeffs[0]换算得到值coeff，然后将coeff赋值给系数矩阵中的每个系数。

HEVCDSPContext -> idct_4x4_luma()

HEVCDSPContext -> idct_4x4_luma()指向处理Intra4x4的CU的DST反变换。相比于视频编码中常见的DCT反变换，DST反变换算是一种比较特殊的变换。4x4DST反变换的C语言版本函数是transform_4x4_luma_8()，它的定义如下所示。

```cpp
#define SCALE(dst, x) (dst) = av_clip_int16(((x) + add) >> shift)

/*
 * 4x4DST
 *
 *      | 29  55  74  84 |
 * H =  | 74  74   0 -74 |
 *      | 84 -29 -74  55 |
 *      | 55 -84  74 -29 |
 *
 */
#define TR_4x4_LUMA(dst, src, step, assign)                        \
    do {                                                           \
        int c0 = src[0 * step] + src[2 * step];                    \
        int c1 = src[2 * step] + src[3 * step];                    \
        int c2 = src[0 * step] - src[3 * step];                    \
        int c3 = 74 * src[1 * step];                               \
                                                                   \
        assign(dst[2 * step], 74 * (src[0 * step] -                \
                                    src[2 * step] +                \
                                    src[3 * step]));               \
        assign(dst[0 * step], 29 * c0 + 55 * c1 + c3);             \
        assign(dst[1 * step], 55 * c2 - 29 * c1 + c3);             \
        assign(dst[3 * step], 55 * c0 + 29 * c2 - c3);             \
    } while (0)

//4x4DST
static void FUNC(transform_4x4_luma)(int16_t *coeffs)
{
    int i;
    int shift    = 7;
    int add      = 1 << (shift - 1);
    int16_t *src = coeffs;

    for (i = 0; i < 4; i++) {
        TR_4x4_LUMA(src, src, 4, SCALE);
        src++;
    }

    shift = 20 - BIT_DEPTH;
    add   = 1 << (shift - 1);
    for (i = 0; i < 4; i++) {
        TR_4x4_LUMA(coeffs, coeffs, 1, SCALE);
        coeffs += 4;
    }
}

#undef TR_4x4_LUMA
```

从源代码可以看出，transform_4x4_luma_8()调用TR_4x4_LUMA()完成了4x4DST的工作。

HEVCDSPContext -> transform_add[4]()

HEVCDSPContext -> transform_add[4]()指向了叠加残差数据的汇编函数。这些函数用于将残差像素数据叠加到预测像素数据上，形成最后的解码图像数据。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体C语言版本处理函数为：

transform_add4x4_8()——4x4块；

transform_add8x8_8()——8x8块；

transform_add16x16_8()——16x16块；

transform_add32x32_8()——32x32块；

这四个函数的定义如下所示。

```cpp
//叠加4x4方块的残差数据
static void FUNC(transform_add4x4)(uint8_t *_dst, int16_t *coeffs,
                                   ptrdiff_t stride)
{
    //最后一个参数为4
    FUNC(transquant_bypass)(_dst, coeffs, stride, 4);
}
//叠加8x8方块的残差数据
static void FUNC(transform_add8x8)(uint8_t *_dst, int16_t *coeffs,
                                   ptrdiff_t stride)
{
    //最后一个参数为8
    FUNC(transquant_bypass)(_dst, coeffs, stride, 8);
}
//叠加16x16方块的残差数据
static void FUNC(transform_add16x16)(uint8_t *_dst, int16_t *coeffs,
                                     ptrdiff_t stride)
{
    //最后一个参数为16
    FUNC(transquant_bypass)(_dst, coeffs, stride, 16);
}
//叠加32x32方块的残差数据
static void FUNC(transform_add32x32)(uint8_t *_dst, int16_t *coeffs,
                                     ptrdiff_t stride)
{
    //最后一个参数为32
    FUNC(transquant_bypass)(_dst, coeffs, stride, 32);
}
```

从源代码可以看出，transform_add4x4_8()、transform_add8x8_8()等函数内部都调用了同样一个函数transquant_bypass_8()，它们的不同在于传递给transquant_bypass_8()的最后一个参数size的值不同。

transquant_bypass_8()

transquant_bypass_8()完成了残差像素数据叠加的工作。该函数的定义如下所示。

```cpp
//叠加残差数据，transquant_bypass_8()
static av_always_inline void FUNC(transquant_bypass)(uint8_t *_dst, int16_t *coeffs,
                                                     ptrdiff_t stride, int size)
{
    int x, y;
    pixel *dst = (pixel *)_dst;

    stride /= sizeof(pixel);
    //逐个叠加每个点
    for (y = 0; y < size; y++) {
        for (x = 0; x < size; x++) {
            dst[x] = av_clip_pixel(dst[x] + *coeffs);//叠加，av_clip_pixel()用于限幅。处理的数据一直存储于dst
            coeffs++;
        }
        dst += stride;
    }
}
```

从源代码中可以看出，transquant_bypass_8()将残差数据coeff依次叠加到了预测数据dst之上。

至此有关IDCT方面的源代码就基本分析完毕了。

雷霄骅
leixiaohua1020@126.com
http://blog.csdn.net/leixiaohua1020

文章标签： FFmpeg libavcodec HEVC 解码器 TU

个人分类： FFMPEG

所属专栏： FFmpeg