Application Report
SPRA805 - June 2002

Variable-Length Decoding on the TMS320C6000 DSP Platform

Oliver Sohm   TMS320C6000 Software Applications

ABSTRACT

The TMS320C62x™ and the TMS320C64x™ DSPs are well suited for high-performance applications such as image and video decoding. Efficient implementation of core decoding steps such as inverse discrete cosine transform (DCT), variable-length decoding, dequantization, and motion compensation is essential for achieving the best possible performance. This application report focuses on optimized implementations of variable-length decoding for the standards MPEG-2 (ISO/IEC 11172 Part 2: Video), ITU-T H.263, and JPEG[1–3]. Ready-to-use solutions are presented as well as an in-depth treatment of optimization techniques.

Optimization of variable-length decoding (VLD) routines can be difficult, because of their inherently serial data flow which makes exploiting parallelism on VLIW-type architectures more challenging. Techniques for optimization and robustness are discussed including bitstream buffer management, look-up table organization, handling of irregular loops, error behavior and exploiting parallelism across multiple decoding steps by integrating VLD with run-length expansion, inverse zigzag, dequantization, saturation and mismatch control. By applying these optimization techniques very good performance can be achieved. For instance, the performance of the MPEG-2 VLD presented here is 10 cycles per decoded symbol plus an overhead of 37 cycles per decoded block for non-intra coded macroblocks. For decoding a 4 Mbit/s MPEG-2 bitstream and assuming 5 bits per coded symbol, the processor load is only about 1.7% on a 600 MHz C64x. This level of performance combined with the flexibility of a software implementation can be a significant advantage over VLD co-processors.

Source code of implementations and examples are available in the attached zip archive. The latest versions of the MPEG-2 VLD routines can be found in the *TMS320C62x Image/Video Processing Library* and *TMS320C64x Image/Video Processing Library*.

Contents

List of Figures

List of Tables

1 Introduction

Variable-length decoding (VLD) is typically one of the most time consuming functions of an image/video decoder, and has therefore been usually implemented in hardware. This application report discusses how VLD routines for JPEG, H.263 and MPEG-2 can be very efficiently implemented in software on the TMS320C6000™. Due to the serial nature of the VLD, exploiting parallelism on VLIW-type processors can be challenging. However, good performance can be achieved through careful optimization of the algorithm and integrating it with other decoding functions such as inverse zigzag, dequantization, saturation and mismatch control. For instance, one MPEG-2 non-intra coded macroblock containing 120 coded symbols can be decoded in 1454 cycles on the C64x™. Note that the routine fully decodes all coefficients which can then be directly fed into the inverse DCT stage. For decoding a 4 Mbit/s MPEG-2 bitstream and assuming 5 bits per coded symbol, the processor load is only about 1.7% on a 600-MHz C64x device. This level of performance, combined with the flexibility of a software implementation, can be a significant advantage over VLD co-processors.

This section first describes the variable-length decoding process that is commonly used in image and video compression standards and then identifies common implementation problems. The next three sections then present detailed implementation examples for an MPEG-2, H.263, and JPEG VLD routine. The application report finishes with a summary and review of the optimization techniques used in the examples.

Most standards, such as JPEG, MPEG and H.263 employ the coding procedure shown in Figure 1.

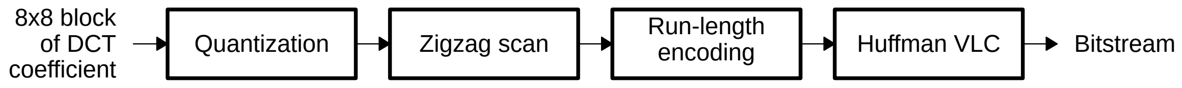


Figure 1. Typical Image/Video Compression System

After run-length encoding, we obtain a pair of values, the number of consecutive zeros (RUN) before the next non-zero quantized DCT coefficient and the magnitude (LEVEL) of that coefficient. Each value pair (also called EVENT) is then assigned a variable-length code word which is obtained through table look-up. The look-up tables (LUT) are specified by the standard, although JPEG also allows user-specified tables. Table 1 shows an extraction of an MPEG-2 VLC table. The attached s denotes the sign of the LEVEL.

TMS320C6000 and C64x are trademarks of Texas Instruments.

Table 1. Example of MPEG-2 VLC Codes and Corresponding RUN-LEVEL Pairs

VLC	Run	Level
10s	0	1
010s	1	1
110s	0	2
0111s	0	3
00101s	2	1
00110s	1	2
11101s	0	5
0000100s	7	1
0000111s	2	2
1111000s	9	1
1111100s	0	9
00100000s	1	5
00100001s	11	1
11111011s	0	13
11111111s	0	15
000000100s	5	2
0000001100s	2	4
0000001101s	16	1
0000000011011s	26	1
0000000011111s	22	1
00000000010000s	0	31
00000000011111s	0	16
000000000010000s	0	40
0000000000010000s	1	18
0000000000010001s	1	17

For less frequently occurring RUN-LEVEL pairs that are not given in the table, the values for RUN and LEVEL are directly inserted in the bitstream, preceded by a special ESCAPE VLC code.

The decoding process follows the reverse flow of the flow shown in Figure 1. A generalized Huffman variable-length decoding (VLD) procedure is shown in Figure 2. First, the length of the VLC code has to be determined, and the bitstream advanced to the next code. Then RUN and LEVEL are obtained through table look-up. The process stops when all symbols of a block have been decoded. The end of a block is typically marked through a special end-of-block (EOB) code.

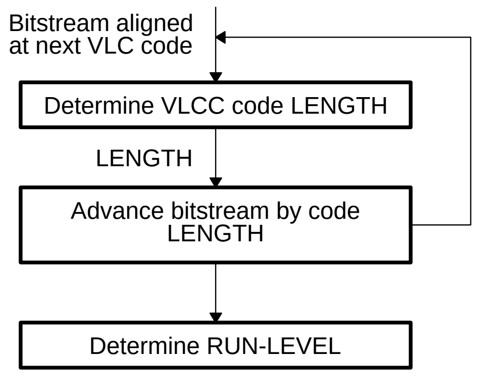

Bitstream aligned at next VLC code
Determine VLCC code LENGTH
LENGTH
Advance bitstream by code LENGTH
Determine RUN-LEVEL

Figure 2. Huffman VLD Procedure

The procedure is not a typical DSP task, and therefore makes implementation difficult. The amount of instruction-level parallelism that can be exploited is limited due to the serial nature of the decoding loop. That is, before the next VLC code can be decoded, the length of the previous one must first have been determined and the bitstream advanced accordingly. This constitutes a recurrence path in the loop which limits the minimum initiation interval (II) that can be achieved by software pipelining. Furthermore, the number of symbols within a block, and thus the number of iterations, are unknown until an EOB code is detected, i.e., the loop is irregular (WHIILE loop). This further complicates software pipelining, since additional instructions and registers are required to preserve live-out variables.

As can be seen in Figure 2, the length of the recurrence path is determined by how many cycles are required for computing the length of the VLC code, and for advancing the bitstream. It is therefore essential for the overall performance of the VLD function to keep these two tasks as short as possible. Determining the length of the VLC code is commonly done by table look-up. The table layout and the indexing scheme chosen, as well as the way the bitstream buffer is managed affects the length of the recurrence path.

The actual RUN and LEVEL values are determined again through look-up tables. The most basic indexing scheme would be to use the VLC code as index. However, if the longest VLC code is, for instance 16 as it is the case for MPEG-2, the table would have 2^{16} = 65,536 entries. This is excessively large and not practical for most implementations. Therefore, the look-up tables have to be compressed by reducing the redundancy contained in Huffman code words. One common way to achieve this is create several sub-tables which group the VLC codes according to their number of leading zeros or ones.

Examples for how look-up tables may be organized, and bitstream buffers may be managed, are given in the next three sections, which discuss VLD implementations for MPEG-2, H.263 and JPEG. Also shown is how performance can be further increased by integrating run-length decoding, inverse zigzag, and dequantization into the VLD loop. This is possible since (due to the serial nature of VLD) many instruction slots remain unused.

The implementations discussed in the following sections are for illustrative purposes, and do not necessarily make use of all possible optimizations. For readers who are interested in only one particular implementation, it is noted that each section is self-contained, and can be followed without reading other sections. Readers who wish to implement a different variable-length decoding routine not specifically discussed in this application report, are referred to section 6, which reviews the implementation techniques used in the various implementations. Also, the section covering MPEG-2 Variable-Length Decoding (section 2) goes into more detail about trade-offs and implementation issues that arise, and how they can be addressed.

2 MPEG-2 Variable-Length Decoding

2.1 Decoding Process

In MPEG-2 [1], the decoding process is macroblock (MB)-based. In 4:2:0 format, one MB consists of 4 luminance block and 2 chrominance blocks. Variable-length decoding (VLD) is followed by run-length expansion, inverse scan (inverse zigzag), dequantization, saturation, and mismatch control. Five different VLC tables from MPEG-2 (B-12 to B-16), are used. With exception of the DC coefficient of an intra block, the decoding process is independent of whether it is a luminance or a chrominance block. Non-intra blocks always use Table B-14 (for the DC coefficient of a non-intra block, one entry in Table B-14 is modified). AC coefficients of intra blocks are decoded using Table B-14 if intra_vlc_format is 0, and Table B-15 if intra_vlc_format is 1 (MPEG-1 uses Table B-14 only). DC coefficients of intra blocks use Table B-12 for luminance blocks, and B-13 for chrominance blocks. Coefficients (excluding the intra DC) that are not given in Table B-14 or B-15. They are coded as escape sequence, using the fixed-length codes given in Table B-16.

2.1.1 Decoding and Dequantization of Intra DC

Rather than coding the DC intra coefficient directly, the difference of two consecutive DC intra coefficients is coded. Differences are only taken within the same block type (i.e., luminance (0), chrominance (1) and chrominance (2)). Therefore, for reconstruction of a DC intra coefficient, first the differential DC is decoded and then added to the previous reconstructed DC. Codes for differential DCs consist of two components. The first one, dc_dct_size, is a variable-length code, which denotes the size (length in bits) of the second component. The second component is a fixed-length word that contains the value used to reconstruct the differential DC value (dct_diff). The algorithm to reconstruct the DC intra coefficient is described by the following pseudo C code [1]:

```
if ( dc_dct_size == 0 ) {
 dct_diff = 0;
} else {
 half_range = 2 ^ ( dc_dct_size - 1 ); Note: ^ denotes power (not XOR)
 if ( dc_dct_differential >= half_range )
  dct_diff = dc_dct_differential;
 else
  dct_diff = (dc_dct_differential + 1) - (2 * half_range);
}
QFS[0] = dc_dct_pred[cc] + dct_diff;
dc_dct_pred[cc] = QFS[0]
```

The obtained DC intra value is then dequantized by multiplying it with a factor which depends on intra_dc_precision. This can be realized by left-shifting by (3-intra_dc_precision).

2.1.2 Decoding and Dequantization of Other Coefficients

Codes for coefficients other than the DC intra, either consist of one single variable-length code denoting RUN and LEVEL (VLC Table B-14 and Table B-15), or of an escape code sequence, in which case RUN is contained in a 6-bit unsigned fixed-length code word, and LEVEL is contained in a 12-bit signed fixed-length code word (Table B-16). The escape code itself is part of the VLC tables. The last bit of a VLC code determines the sign of LEVEL (a last bit of 0 indicates a positive, and last bit of 1, a negative LEVEL).

The value of RUN indicates how many consecutive coefficients (from the last decoded coefficient) shall be set to zero. The subsequent coefficient is then set to the signed LEVEL. If an End of Block (EOB) code was decoded, then all remaining coefficients in the block are set to zero.

The decoded block now contains 64 coefficients arranged in a zigzag scanning order. The inverse scan rearranges the coefficients such that a “normal” raster scan order is obtained. The parameter alternate_scan defines which one of two possible zigzag scanning orders is to be used.

The dequantized coefficient is obtained according to the following formula:

DQ_COEFF[x][y] = ((2 * COEFF[x][y] + k) * W[x][y] * quantiser_scale)/32

where

intra blocks: k = 0

non-intra blocks: k = 0 for COEFF[x][y] = 0, k = 1 for for COEFF[x][y] > 0 and k = –1 for COEFF[x][y] < 0.

For 4:2:0 format, a different weighting matrix W is used for intra and non-intra blocks. For 4:2:2 and 4:4:4 formats, there are four weighting matrices, depending on which combination of block type (intra/non-intra and luminance/chrominance) is used. A quantiser_scale factor is used, which depends on the quantiser_scale_code and q_scale_type.

The dequantized coefficients are then saturated to lie in the range [–2048:+2047].

Finally, mismatch control is performed. If the sum of all reconstructed and saturated coefficients is even (LSB = 0), then the LSB of the last coefficient is toggled.

For non-coded blocks, i.e., blocks whose corresponding bit in coded_block_pattern is zero, all pixels values shall be zero.

2.2 Possible Ways of Implementation

The type of VLD performed depends on intra_vlc_format and whether the macroblock is intra or non-intra. The following cases have to be considered:

- Intra block using Table B–14, first coefficient (intra DC) requires different decoding procedure
- Intra block using Table 2, first coefficient (intra DC) requires different decoding procedure
- Non-intra block using Table B–14, first coefficient requires modification of table

Either 3 different functions can be implemented, one for each case, or if this is not desirable because of code size, 2 different functions can be implemented.

1. Combining all three cases into one function may not be desirable because the differences between the cases are such that the implementation would be less efficient. For instance, for non-intra blocks this would require skipping intra DC decoding. This would be a simple predicated branch, but would require that only operations for intra DC decoding be skipped which means that other operations could not be issued in parallel, and so resource utilization is lower. However, it can be seen in all 3 cases that the first coefficient of a block requires different handling than the remaining AC coefficients. If decoding of the AC coefficients is the same procedure for intra and non-intra blocks, and just requires using a different VLC table. Then, for each block:

   ```
   if (intra)
      decode first coefficient intra
   else
      decode first coefficient non-intra
   while (!eob)
      decode AC coefficient
   ```

   The while loop can be software pipelined and the prolog/epilog be merged with first coefficient decoding and other outer loop code. The merged of code would, however, have to be different for both cases of intra and non-intra. This is actually not much different than having two functions, one for intra and one for non-intra blocks, with the only difference that two branches are required:

   ```
            <setup code>
   block:
   [!intra] B non-intra
            <intra DC decoding merged with prolog and outer loop code>
   [intra]  B AC_loop
   non_intra:
           <non-intra DC decoding merged with prolog and outer loop code>
   AC_loop:
           <AC decoding kernel>
           B block

           <exit code>
   ```

   If Table B-14 and Table B-15 require different decoding methods (due to the different structure of the tables), then we also would have two different AC_loops. In that case, two different functions would be more appropriate.

2. There are two possibilities if the VLD is split into two separate functions:
   a. One function for intra and another for non-intra blocks
   b. One function for Table B–14, and another Table B-15.

For possibility “b” with Table B–14, this would look like combining all three cases as described above, since this table is used for intra and non-intra. For Table B-15 only, non-intra decoding is required.

Another consideration is the effect on the performance. By far more blocks are likely to be non-intra coded, and a special implementation, which is optimized for this case may help reduce the overall decoding time. Also, commonly a separate function for intra and non-intra is used in implementations such as the one by the MPEG Software Simulation Group. Therefore, it was chosen to implement one function for intra and another for non-intra blocks.

2.3 Bitstream Management

This section explains how the bitstream is buffered in registers and how to initialize those registers. To reduce the delay caused by updating the buffer from memory, a double-buffering scheme is used, as shown in Figure 3.

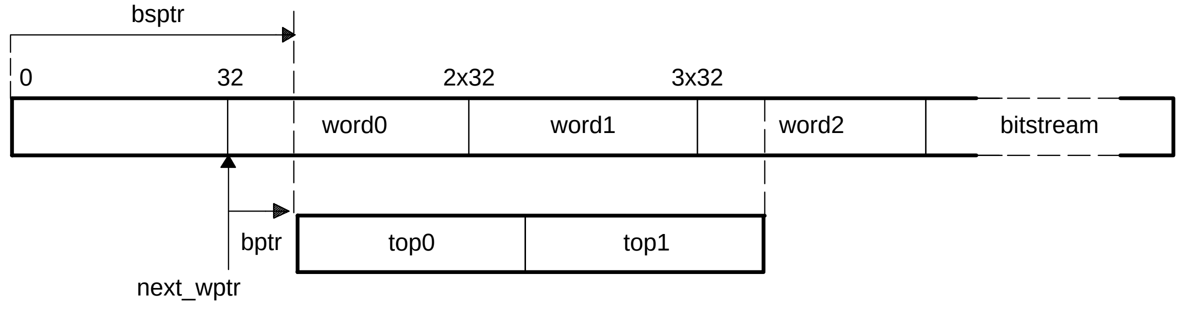


Figure 3. Bitstream Buffering

The variable bsptr points to the position of the current bit in the bitstream, buffer bsbuf[ ], which is assumed to be stored in memory in 32-bit elements. Since bsptr may not be a multiple of 32, an index, next_wptr, is used and set to the next lower multiple of 32 by dividing bsptr by 32. For instance, if bsptr is 40, this yields next_wptr = 1.

```
next_wptr = (bsptr >> 5);
```

The variable bptr is the bit pointer which points to the current bit *within* the word pointed to by next_wptr:

```
bptr = bsptr & 31;
```

Now the next three words are read from the bitstream buffer bsbuf[ ] which is accessed in a circular mode. The buffer is assumed here to have a size of 64 words:

```
word0 = bsbuf[next_wptr];
next_wptr = (next_wptr + 1) & 63;
word1 = bsbuf[next_wptr];
next_wptr = (next_wptr + 1) & 63;
word2 = bsbuf[next_wptr];
next_wptr = (next_wptr + 1) & 63;
```

To obtain top0, which contains the first 32 bits left-aligned at the current bit, word0 is shifted left by bptr, such that the current bit becomes the MSB. This creates empty LSBs that have to be filled with the MSBs of word1. To do that, the required MSBs are brought into the same position as the empty LSBs by shifting word1 to the right by 32-bptr (unsigned shift). The result is then copied into top0 through an addition. The same procedure is used to obtain top1. Registers top0 and top1 now contain the first 64 bits of the bitstream:

```
top0 = (word0<<bptr + word1>>(32-bptr));
top1 = (word1<<bptr + word2>>(32-bptr));
```

The bitstream variables are set up now, and the VLD routine can be called. Within the routine, each time after a variable-length code is decoded, the bitstream has to be advanced to the start of the next code. The register top0 can be easily updated from top1, if len contains the length in bits of the code just decoded:

```
top0 = (top0<<len + top1>>(32-len));
```

Next, top1 is updated from word1 and word2 after increasing bptr by len. If necessary (i.e., if the new bptr is greater than 32), word2 needs to be updated from memory first. Also, bptr is always relative to the next lower word boundary, and therefore needs to be ANDed with 31, in case it becomes greater than 32.

```
bptr += len;
if (bptr >= 32) {
bptr = bptr & 31;
word1 = word2;
word2 = bsbuf[next_wptr];
next_wptr = (next_wptr + 1) & 63;
}
top1 = (word1<<bptr + word2>>(32-bptr));
```

2.4 Intra and Non-Intra AC Decoding Loop

The goal of this, and the next section, is to translate the description of the decoding process, as given in the standard, into C source code, which then serves as reference and model for further optimizations. This section discusses decoding of the AC coefficients, and the next section presents decoding procedures for the DC coefficient. This also involves the design of look-up tables, with the aim of finding the right trade-off between low computational complexity for fast decoding, and small table sizes for low memory usage. The AC coefficient decoding loop is the same for intra and non-intra macroblocks, except that in the non-intra case, two additional operations are required for dequantization.

2.4.1 Length Computation

To advance the bitstream to the next code, the length of the current code has to be determined. A look-up table is used to do that. In most cases, the number of leading zeros or ones determines the length of the code. The *norm* instruction of the C6000™ can be used to count the number of redundant 2's complement sign bits (e.g., 0001 xxxx ... xxxx returns 2, 1111 10xx ... xxxx returns 4).

It turns out that for both tables, B–14 and B–15, the maximum number of redundant sign bits is 10. To represent this number, we need at least 4 bits. However, in both tables there is one code which contains only 1's. In the case of B-14, this is code 11, and in case of B–15, it is 111 111, plus a sign bit. If the sign bit is 0, the norm instruction returns 7, in case of B–15. A problem arises when the sign bit is 1. Then, norm cannot find the end of the code and runs into the next code. The worst case is that, assuming a bitstream management as described above, all 32 bits of the register containing the top of the bitstream are 1. In that case, norm returns 31. Therefore, the norm value should be represented by 5 bits instead of 4.

C6000 is a trademark of Texas Instruments.

Since the norm value is ambiguous, as in the cases just mentioned, an additional 4 bits following the redundant sign bits are used to obtain an 9-bit index into the look-up table:

```
n = _norm(top0);
index = (n<<4) + ((top0<<n) >> 28);
```

Note that if there are less than 4 bits after the redundant sign bits in the code, the last bits belong to the next code in the bitstream. When building a look-up table, these unused bit positions have to be treated as “don’t cares”, and expanded into individual table entries which all have the same length value (e.g., see VLCs 10 and 110 in Table 1). That way, the bits that are read beyond the end of the current code do not matter.

Table 2. MPEG-2 Table B-15 VLC Codes

VLC	Sign Bit	Run	Level	Len	NORM	VLC<< NORM	NORM (bin)	4 Extra Bits
10	S	0	1	3	0	10	00000	10xx
010	S	1	1	4	0	010	00000	010x
0110	(End of Block)			4	0	0110	00000	0110
0111	S	0	3	5	0	0111	00000	0111
110	S	0	2	4	1	10	00001	10xx
0010 1	S	2	1	6	1	0101	00001	0101
0011 0	S	1	2	6	1	0110	00001	0110
0011 1	S	3	1	6	1	0111	00001	0111
0010 0000	S	1	5	9	1	0100 000	00001	0100
0010 0001	S	11	1	9	1	0100 001	00001	0100
0010 0010	S	0	11	9	1	0100 010	00001	0100
0010 0011	S	0	10	9	1	0100 011	00001	0100
0010 0100	S	13	1	9	1	0100 100	00001	0100
0010 0101	S	12	1	9	1	0100 101	00001	0100
0010 0110	S	3	2	9	1	0100 110	00001	0100
0010 0111	S	1	4	9	1	0100 111	00001	0100

Table 2 shows the VLC codes of MPEG-2, Table B–15, which have a *norm* of 0 and 1. The last two columns, “NORM (bin)” and “4 extra bits”, form the 9-bit index into the length look-up table. For instance, to distinguish between the codes 0110 (4 bits long) and 0111s (5 bits long), which both have a norm of 0, at least 4 extra bits are necessary. Since the first 7 bits of the index are identical, only the 8th bit distinguishes the two codes. The index for the first code in Table 2 (10s) has two “don’t care” bits, i.e., four table entries containing a length value of 3 have to be created: 008h, 009h, 00Ah and 00Bh.

To cover the special cases of 111 1111 and 11, these codes have to be replicated in the table for all possible norm values they can cause. Helpful is that for these two codes, no matter how many redundant sign bits are counted, the 4 extra bits are always 1xxx, since we are always left with at least one 1 bit after VLC<<NORM. Note that, in case norm overruns into the next VLC, this remaining bit is not even part of the current VLC, which does not matter. These two codes can be unambiguously decoded because, for all possible norm values they can cause, 1xxxx is unique.

The length look-up table has 2^9 = 512 entries. If the length is stored as an 8-bit number (e.g., unsigned character in C), then the size of the table is 512 bytes. The number of entries that are actually used is 138.

To advance the bitstream, we also need the value (32-len). This could be easily calculated, but as we will see later, the length computation is on the critical recurrence path. We could save the operation (32-len) by looking it up in a table simultaneous with looking up len. This method works fine on the C62x™ architecture. However, on C64x, cross path delay slots actually lengthen the path instead of shortening it when this method is used. Therefore, other techniques have to be used, which will described in more detail later.

2.4.2 Run-Level Decoding

Again a look-up table is used to perform run-level decoding. Two methods are presented in the following sections. One method looks up the absolute level value and extracts the sign bit from the code word; the other one incorporates both positive and negative levels into the look-up table.

2.4.2.1 Computing Signed Level

The run-level look-up table uses a similar indexing as the length table. In case of the length table, many VLC codes had the same length, and an 9-bit index was sufficient to unambiguously assign the length to a VLC code. However, run-level values are unique for each and every code. By closely examining the VLC table, it can be seen that the length of a VLC, which was previously computed, combined with the 4 right-most bits (excluding sign bit) of the code word, is sufficient to distinguish between all codes.

Table 3 shows the codes of Table B–15 up to a length of 7. Note that the 4 right-most bits are aligned at the left-most bit of the code if the length of the code is less than 4. Len and the 4 right-most bits form the 9-bit table index. Both run and level can be stored as unsigned 8-bit value.

C62x are trademarks of Texas Instruments.

Table 3. Run-Level Look-Up

VLC	Sign	Run	Level	Len	VLC	Len (bin)	4 Right-Most Bits of VLC (w/o sign)
10	S	0	1	3	10	00011	10xx
010	S	1	1	4	010	00100	010x
0110	End of Block	65	0	4	0110	00100	0110
110	S	0	2	4	110	00100	110x
0111	S	0	3	5	0111	00101	0111
00101	S	2	1	6	0010 1	00110	0101
00110	S	1	2	6	0011 0	00110	0110
00111	S	3	1	6	0011 1	00110	0111
11100	S	0	4	6	1110 0	00110	1100
11101	S	0	5	6	1110 1	00110	1101
000100	S	0	7	7	0001 00	00111	0100
000101	S	0	6	7	0001 01	00111	0101
000110	S	4	1	7	0001 10	00111	0110
000111	S	5	1	7	0001 11	00111	0111

The decoding process can then be performed as follows:

```
rld_left = len-5;
if (len<5) rld_left=0;
run_level = rld_table[(len<<4) + _extu(top0_bk, rld_left, 32–4)];
run = run_level >> 8;
level = run_level & 0xFF;
sign = _extu(top0_bk, len–1, 31);
if (sign) level=–level;
```

2.4.2.2 Look-Up Signed Level

The other method is to look up the signed level rather than extracting the sign bit. This requires a larger table (now 10-bit table index) but saves the last two operations in the above code. The table looks like the one above, but now also contains the negative level. The index is now obtained through Len and the 5 right-most bits, where the last bit is the sign bit. The corresponding code looks like this:

```
rld_left = len–5;
if (len<5) rld_left=0;
run_level = rld_table[(len<<5) + _extu(top0_bk, rld_left, 32–5)];
run = run_level >> 8;
level = run_level & 0xFF;
```

The layout of the table in memory is one byte for level, followed by one byte for run. A combined run-level look-up table has the advantage that memory accesses are more localized, which may be beneficial for cached processor architectures. To further improve cache performance, shorter codes can be grouped together, since they are likely to be accessed more frequently than longer codes. This is automatically achieved in the above schemes by basing the index on the length of the code.

2.4.2.3 Special Codes

Escape codes can be detected by their length, which is always 24. In the length look-up table, the escape code is assigned 24. The escape code does not appear in the run-level table, instead the following code is used to decode escape sequences:

```
if (len==24)
{
  run   = _extu(top0,  6, 26);
  level = _ext (top0, 12, 20);
}
else
{
 /* run-level table look-up */
}
```

The end of block (EOB) codes are stored in the look-up tables with a run of 65. Since this is an invalid run (maximum allowed in an 8x8 block is 63), it can be used to detect an EOB and set a flag:

```
eob_err = (run >= 65);
```

Other invalid codes can then be assigned a run value greater than 65. It can then be distinguished whether an erroneous code occurred or an EOB. In case of an error the VLD routine should indicate to the decoder that it has to resynchronize by returning an error flag.

2.4.3 Inverse Zigzag, Dequantization, Saturation and Mismatch Control

In MPEG-2, dequantization is performed as follows:

intra MBs: `out = (2*level                  * W * qscl)/32`

inter MBs: `out = ((2*level + Sign(level)) * W * qscl)/32`

where Sign(x) = –1 for x < 0, 0 for x = 0 and +1 for x > 0. W is the weighting factor from the corresponding position in the weighting matrix, and qscl is the quantiser_scale. The division, /, is an integer division with truncation of the result toward zero. For example, 7/4 and –7/–4 are truncated to 1 and –7/4 and 7/–4 are truncated to –1 [1].

First, the sign is determined and added to 2*level in case of inter macroblock mode:

```
neg = (level < 0);
pos = (level > 0);

f1 = 2*level;

if (inter_block)
{
 f1 = f1 – neg;
 f1 = f1 + pos;
}
```

Now the quantization matrix element is found at the zigzag position, and is multiplied with qscl and 2*level. Note that to be able to access the weighting matrix through the accumulated run, the matrix has to be stored in zigzag order.

```
W = *(Wptr += run + 1);
f3 = f1 * qscl * W;
```

For negative numbers, 31 needs to be added before dividing by 32, to achieve truncation towards zero, as required by the standard. Saturation to a signed 12-bit word can be achieved through the C6000 instruction, SSHL, which shifts left by a given amount and saturates the result to 32 bits. To divide by 32, we have to shift the result right by 20 (5 + 15).

```
if (neg) f3 += 31;
f5 = _sshl(f3, 15) >> 20;
```

For mismatch control, it has to be determined if the sum of all coefficients is odd or even:

```
sum ^= f5;
```

Finally, the un-zigzagged position of the DCT coefficient is determined, and the coefficient stored:

```
cnum = *(zzptr += run + 1);
out[cnum] = f5;
```

This completes the decoding process of one symbol. The loop exits after all symbols of a block have been decoded or an error has occurred. After the loop, the last coefficient of the previously decoded block is modified by toggling the last bit in accordance with mismatch control:

```
if ((sum&1)==0)
{
    out[63] ^= 1;
}
```

2.5 First Coefficients

The first coefficient of a block requires special treatment which is described in this section.

2.5.1 Intra Blocks

For the coding of intra DC coefficients, a VLC code is used to specify dc_size, which is the length (in bits) of the fixed-length code that follows for the DC differential. To decode the intra DC, we have to determine dc_size and the length of the VLC code. Different code tables are used for luminance and chrominance blocks. Fundamentally, two different approaches are possible. A look-up table can be used similar to the scheme described for decoding the AC coefficients, or a common rule can be identified, which relates the VLC code to its length and the dc_size value. The latter approach shall be presented here.

It was found that the number of leading ones of the code is well correlated to the length and dc_size. The number of leading ones can be easily found using the C6000 instruction, LMBD. Except for a few cases, the length is given by LMBD + 1, the dc_size value is given by LMBD + 2 for luminance, and LMBD + 1 for chrominance.

To distinguish between luminance and chrominance blocks, a value cc has to be determined, where 0 denotes luminance Y, 1 denotes chrominance Cr and 2 denotes chrominance Cb. It can be derived from the block number (0..5), as follows (see Table 4):

```
cc = (block<4) ? 0 : (block&1)+1.
```

Table 4. Intra DC Decoding for Luminance Blocks (cc = 0, Table B–12)

VLC	dc_size	LEN	LMBD(0)	Add to LMBD to get LEN		Add to LMBD to get dc_size	
100	0	3	1	2	if (lmbd<2)	–1	If 100, dct_size = 3
00	1	2	0	2	if (lmbd<2)	1	If 00, dct_size = 1
01	2	2	0	2	if (lmbd<2)	2	
101	3	3	1	2	if (lmbd<2)	2	
110	4	3	2	1		2	
1110	5	4	3	1		2	
11110	6	5	4	1		2	
111110	7	6	5	1		2	
1111110	8	7	6	1		2	
11111110	9	8	7	1		2	
111111110	10	9	8	1		2	
111111111	11	9	9	0	if (lmbd=>9)	2	

Table 4 and Table 5 show how, in each case, dc_size and len can be derived from the LMBD(0,top0) value of the code word. Note that the last VLC codes in each table consist of 1's only, i.e., there is no bit change. Therefore, LMBD may actually return a greater value than 9 for luminance, or 10 for chrominance blocks, because the bits of the next VLC codes are counted until a 0 is detected.

Table 5. Intra DC Decoding for Chrominance Blocks (cc=1 and cc=2, Table B–13)

VLC	dc_size	LEN	LMBD(0)	Add to LMBD to get LEN		Add to LMBD to get dc_size	
00	0	2	0	2	if (lmbd==0)	0	If 00, dct_size = 1
01	1	2	0	2	if (lmbd==0)	1	
10	2	2	1	1		1	
110	3	3	2	1		1	
1110	4	4	3	1		1	
11110	5	5	4	1		1	
111110	6	6	5	1		1	
1111110	7	7	6	1		1	
11111110	8	8	7	1		1	
111111110	9	9	8	1		1	
1111111110	10	10	9	1		1	
1111111111	11	10	10	0	if (lmbd=>10)	1	

The decoding process can be written in pseudo C code as follows:

```
b = _lmbd(0, top0);

len = b + 1;

if (cc==0)                          /* luminance block */
{
    dc_size = b + 2;

    if (top0>>29 = ‘100’) dc_size-=3; /* special case of VLC=100 */
    if (top0>>30 = ‘00’)  dc_size−−;  /* special case of VLC=00  */
    if (b<2) len++;                   /* special case of LMBD<2  */
    if  (b>=9)                        /* last VLC no final 0!    */
    {
      len=9;
      dc_size=11;
    }
}
else                                /* chrominance block */
{
    dc_size = b + 1;

    if (top0>>30 = ‘00’)  dc_size−−;  /* special case of VLC=00  */
    if (b==0) len++;                  /* special case of LMBD=0  */
    if (b>=10)                        /* last VLC no final 0!    */
    {
      len=10;
      dc_size=11;
    }
}
```

This appears to be complicated code, but can be realized in assembly with 4 cycles plus one cycle for setting up registers on parallel paths.

```
            MVK   .S1     11,           A7_const11
||          MVK   .S2     11,           B7_const11
||          MV    .L2X    A8_top0,      B8_top0

            LMBD  .L1     0,            A8_top0,  A2_b        ; leading 1’s
||          LMBD  .L2     0,            B8_top0,  B9_b        ; leading 1’s
||          ZERO  .D1     A1_cc0                              ; cc0=0
||          SHRU  .S1     A8_top0,      29,       A10_d1      ; 100 special case
||          SHRU  .S2X    A8_top0,      30,       B2_c        ; 00  special case
||          MV    .D2     B7_const11,   B1_d                  ; anything !=0
```

```
            ADD   .D1    A2_b,         1,        A5_len      ; general len
||[!B0_cc]  ADD   .S2    B9_b,         2,        B6_dc_size  ; lum: dc_size
||[!B0_cc]  CMPGT .L1    A2_b,         8,        A1_cc0      ; lum: lmbd>=9
||[!B0_cc]  SHRU  .S1    A2_b,         1,        A2_b        ; lum: lmbd<2
||[!B0_cc]  SUB   .L2X   A10_d1,       4,        B1_d        ; lum: 100 special case
||[!B0_cc]  MPY   .M2    0,            B1_cc1,   B1_cc1      ; lum: cc1=0
|| [B0_cc]  ADD   .D2    B9_b,         1,        B6_dc_size  ; chrom: dc_size

   [B0_cc]  CMPGT .L2X   A2_b,         9,        B1_cc1      ; chrom: lmbd>=10
|| [!A2_b]  ADD   .S1    A5_len,       1,        A5_len      ; lum: lmbd<2, chrom: lmbd=0
|| [!B1_d]  SUB   .L2    B6_dc_size,   3,        B6_dc_size  ; lum: 100 special case
|| [!B2_c]  SUB   .D2    B6_dc_size,   1,        B6_dc_size  ; lum+chrom: 00 special case

  [A1_cc0]  SUB   .S1    A7_const11,   2,        A5_len      ; lum:   len=9
||[A1_cc0]  MV    .S2    B7_const11,             B6_dc_size  ; lum:   dc_size=11
||[B1_cc1]  MV    .L1    A7_const11,   1,        A5_len      ; chrom: len=10
||[B1_cc1]  MV    .L2    B7_const11,             B6_dc_size  ; chrom: dc_size=11
```

Using the table look-up method on the other hand requires a few cycles to calculate an index plus 5 cycles for the load instructions to return len and dc_size. One could argue that,, by using the table look-up method other instructions can be scheduled in the delay slots of the load instructions. However, typically, until one knows len and dc_size no other tasks can be performed in any case. Thus, this method saves some cycles and eliminates the need for extra look-up tables.

Once the length of the VLC code and dc_size have been determined, the coded DC differential can be extracted from the bitstream though dc_diff = (top0 << len) >> (32–dc_size). As specified in the MPEG-2 standard, the final DC coefficient outi[0] is then obtained as follows.

```
dc_diff=0;
t1 = top0 << len;
if (dc_size!=0)
{
    dc_diff = t1 >> (32-dc_size);
    half_range = 1 << (dc_size–1);
    if (dc_diff < half_range)
        dc_diff = (dc_diff+1)–(2*half_range);
}
val = (dc_pred[cc]+= dc_diff);

outi[0] = val << (3-intra_dc_precision);
```

2.5.2 Non-Intra Blocks

First coefficients of non-intra block are decoded as all other coefficients in the block (using Table B–14 only), except for one special case. If a run of 0 and a level of 1 occurs as first coefficient of a block, then the code, 1s, is used instead of 11s. Since the EOB code, 10, cannot occur as the first code of a block, the 1s code cannot be mistaken as an EOB. To detect this special case, one needs only to check the first bit of the first code of the block. If it is 1, then it can only be the code 1s, since all other codes start with 0. In this case, we can skip the decoding process, set len to 2, run to 0 and level to 1, and directly proceed to dequantization.

2.6 C Model Code

The following C-model code uses table look-ups for both the len and 32-len (denoted as len_c). To decrease overhead, the routine processes all 6 blocks in an MPEG-2 macroblock (assuming 4:2:0 video format). This requires additional checking of the coded block pattern (cpb) information in the MB header. If a block is marked as not coded, the VLD process for that block has to be skipped. This is achieved as follows:

```
for (block=0; block<6; block++)
{

/* ---------------------------------------------------------------------
 *  cbp bit:  5 4 3 2 1 0
 *  block:    0 1 2 3 4 5
 *  If a bit in the cbp field is zero, the corresponding block
 *  is not coded.
 * ---------------------------------------------------------------------*/
    if (!(cbp & (1 << (5–block))))
        continue;

    < VLD routine >

 }
```

Note that C6000 DSPs handle left shifts (<<) differently than PC and UNIX platforms. The VLD routines makes use of a shift value of 32, i.e., x<<32. The C6000 returns 0, while PC and UNIX platforms return the original value x [5]. To ensure that the routines work correctly across all the platforms, the following macros should be used:

```
#ifdef _TMS320C6X
# define SHL(x,y) ( (x) << (y) )
# define SHR(x,y) ( (x) >> (y) )
#else
# define SHL(x,y) ( ((y) & 32) ? 0 : ((x) << (y)) )
# define SHR(x,y) ( ((y) & 32) ? 0 : ((x) >> (y)) )
#endif
```

Other instructions such as LMBD and EXT/EXTU have to be simulated if the code is to run on other than the C6000 platforms.

The intra VLD routine decodes all coefficients of an MPEG-2 intra coded macroblock as described previously. It checks coded block pattern (cbp), performs DC and AC decoding, including variable-length decode, run-length expansion, inverse zigzag, dequantization, saturation, and mismatch control. Similarly, the inter VLD routine decodes all coefficients of an MPEG-2 non-intra coded macroblock. The routines have the following C prototypes:

```
void ti_mpeg2_vld_intra(const short *Wptr,
                        short *outi,
                        unsigned int *Mpeg2v,
                        int   dc_pred[3]);

void ti_mpeg2_vld_inter(const short *Wptr,
                        short *outi,
                        unsigned int *Mpeg2v);
```

Wptr: Pointer to the quantization matrix. Wptr[ ] must be in zigzag scan order (provided in ti_quant_zz.h), since dequantization is performed before the inverse zigzag scan.

outi: Pointer to the output array where the decoded and dequantized IDCT coefficients are stored. The array must have a size of 6 * 64 elements. All elements in the array must be set to zero before the routine is called.

Mpeg2v: Pointer to the bitstream context structure which includes the input buffer variables. The structure Mpeg2v is defined as follows:

```
typedef struct {
unsigned int  *bsbuf;      // pointer to bitstream buffer
unsigned int  next_wptr;   // next word to read from buffer
unsigned int  bptr;        // bit position within word
unsigned int  word1;       // word aligned buffer
unsigned int  word2;       // word aligned buffer
unsigned int  top0;        // top 32 bits of bitstream
unsigned int  top1;        // next 32 bits of bitstream
unsigned char *scan;       // inverse zigzag scan matrix
unsigned int  intravlc;    // intra_vlc_format
unsigned int  quant_scale; // quant_scale
unsigned int  dc_prec      // intra_dc_precision
unsigned int  cbp;         // coded_block_pattern
unsigned int  fault;       // fault condition (returned)
};
```

dc_pred: Intra DC prediction array. The first element of dc_pred is the DC prediction for Y, the second for Cr, and the third for Cb.

The routine sets the fault flag Mpeg2v. fault to 1 if an invalid VLC code was encountered or the total run went beyond 63. In these cases, the decoder has to resynchronize.

Before calling the routines, the bitstream variables in Mpeg2v have to be initialized, as described in section 2.3. Note that the routine returns the updated state of the bitstream buffer variables, top0, top1, word1, word2, bptr, and next_wptr. If all other functions that access the bitstream in a decoder system maintain the buffer variables in the same way, then the above initialization procedure has to be performed only once at the beginning.

See the *TMS320C62x Image/Video Processing Library Programmer's Reference* (SPRU400) or *TMS320C64x Image/Video Processing Library Programmer's Reference* (SPRU023) for more details on how to use these routines.

2.6.1 Intra VLD

```
#define BSBUF_SIZE  512                 /* bitstream buffer size in words   */

#include ”c6000.h”

/* ------------------------------------------------------------------------ */
/*  Run-level tables                                                        */
/* ------------------------------------------------------------------------ */
extern unsigned short rld_table0[576];
extern unsigned short rld_table1[576];

/* ------------------------------------------------------------------------ */
/*  Length tables                                                           */
/* ------------------------------------------------------------------------ */
extern const unsigned char len_c_tbl0[512];
extern const unsigned char len_tbl0[512];
extern const unsigned char len_c_tbl1[512];
extern const unsigned char len_tbl1[512];

/* ------------------------------------------------------------------------ */
/*  Function                                                                */
/* ------------------------------------------------------------------------ */

void mpeg2_vld_intra_cn(const short *Wptr, short *outi, unsigned int *Mpeg2v,
        int dc_pred[3])
{

/* ------------------------------------------------------------------------ */
/*  Get bitstream info.                                                     */
/* ------------------------------------------------------------------------ */
      unsigned int        *bsbuf    = (unsigned int *) Mpeg2v[0];
      int                 next_wptr = Mpeg2v[1];
      int                 bptr      = Mpeg2v[2];
      unsigned int        word1     = Mpeg2v[3];
      unsigned int        word2     = Mpeg2v[4];
      unsigned int        top0      = Mpeg2v[5];
      unsigned int        top1      = Mpeg2v[6];
      const unsigned char *zzptr    = (const unsigned char *) Mpeg2v[7];
      int                 intra_vlc_format = Mpeg2v[8];
      int                 qscl      = Mpeg2v[9];
      int                 intra_dc_precision = Mpeg2v[10];
      int                 cbp       = Mpeg2v[11];

      unsigned int        bptr_cmpl, top0_bk;

/* ------------------------------------------------------------------------ */
/*  Block number (0–3: lum, 4, 5: chrom)                                    */
/* ------------------------------------------------------------------------ */
    int block;

/* ------------------------------------------------------------------------ */
/*  Variables for intra DC decoding                                         */
/* ------------------------------------------------------------------------ */
    unsigned int cc, a_cc0, a_cc1, b, c, d, dc_size;
    int dc_diff, val, half_range;
```

```
/* ---------------------------------------------------------------------- */
/*  Variables for length, run, and level decoding                        */
/* ---------------------------------------------------------------------- */
    unsigned int t1, t2, t4, t4b, t5, t7, t8, t9;
    int rld_left, rld_index;
    unsigned short run_level;
    unsigned char run, len, len_c;
    const unsigned char *t3, *t3b;
    short level;
    unsigned int eob_err, nrm, fault;
    unsigned int test1, test2;

/* ---------------------------------------------------------------------- */
/*  Variables for dequantization                                          */
/* ---------------------------------------------------------------------- */
    int neg, f1, f3, f5, qW, sum;
#ifdef _TMS320C6X
    int f4;
#else
    _int64 f4;
#endif
    const short *Wptr_origin=Wptr;
    short W;
    unsigned char cnum;

    fault = 0;

    for (block=0; block<6; block++)
    {

    /* -------------------------------------------------------------------
     *  cbp: Bit 5 4 3 2 1 0. If the corresponding bit is zero block number,
     *  (5-bit pos) is not coded.                                         */
     * ------------------------------------------------------------------
        if (!(cbp & (1 << (5–block))))
            continue;

        zzptr = (const unsigned char *) Mpeg2v[7];
        Wptr=Wptr_origin;

        cc = (block<4) ? 0 : (block&1)+1;

        sum = 0;
        eob_err = 0;

    /* ------------------------------------------------------------------
     *  Decode first coefficient (DC coefficient) for INTRA block.
     * ------------------------------------------------------------------ */

    /* ------------------------------------------------------------------
     *  Intra DC: Decode dct_size and len (luminance and chrominance).
     * ------------------------------------------------------------------ */
        a_cc0 = a_cc1 = 0;
        b = _lmbd(0, top0);
        c = top0 >> 30;
        d = top0 >> 29;
        len = b + 1;
        if (cc==0)
        {
            dc_size = b + 2;
            a_cc0 = (b>=9);          /* the last VLC does not have a final 0!*/
            b = b >> 1;
            d -= 4;
        }
        else
        {
            dc_size = b + 1;
            a_cc1 = (b>=10);       /* the last VLC does not have a final 0! */
            d = 1;                                          /* anything !=0 */
        }

        if (!b) len++;
        if (!c) dc_size--;
        if (!d) dc_size-=3;
        if (a_cc0)
        {
            len=9;
            dc_size=11;
        }
        if (a_cc1)
        {
            len=10;
            dc_size=11;
        }

    /* --------------------------------------------------------------------
     *  Intra DC: Obtain QFS[0] from dc_size and dc_differential.
     * -------------------------------------------------------------------- */
        dc_diff=0;
        t1 = top0 << len;
        if (dc_size!=0)
        {
            half_range = 1 << (dc_size-1);
            dc_diff = t1 >> (32-dc_size);
            if (dc_diff < half_range)
                dc_diff = (dc_diff+1)-(2*half_range);
        }
        val = (dc_pred[cc]+= dc_diff);


    /* --------------------------------------------------------------------
     *  Intra DC: Dequantization and store result
     * -------------------------------------------------------------------- */
        outi[block*64+0] = val << (3-intra_dc_precision);

    /* --------------------------------------------------------------------
     *  Intra DC: Mismatch control
    /* -------------------------------------------------------------------- */
        sum += outi[block*64+0];
```

```
/* –––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
 *  Intra DC: Now that we know the length of the current VL code, we
 *  can advance bitstream to the next one, i.e., by len+dc_size.
 * ––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––– */
    len+=dc_size;

/* –––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
 *  From here, it is the same code as used in the loop for the other
 *  coefficients.
 * ––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––– */

/* –––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
 *  1. Update top0 from top0 and top.
 * ––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––– */
    t5  = top0 << len;
    t7  = top1 >> (32–len);
    top0_bk = top0;
    top0 = t5 + t7;

/* –––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
 *  2. Update top1 from word1 and word2 after increasing bptr by len.
 *  If neccesary (i.e., if new bptr is greater than 32), update word1
 *  and word2 from memory first. Do not forget that bptr is always
 *  relative to the next lower word boundary, and therefore needs to be
 *  ANDed with 31, in case it become >= 32.
 * ––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––– */
    bptr += len;
    test2 = (bptr >= 32);
    if (test2) {
        word1 = word2;
        word2 = bsbuf[next_wptr];
        next_wptr += 1;
        next_wptr &= (BSBUF_SIZE–1);
    }
    bptr = bptr & 31;
    bptr_cmpl = 32 – bptr;
    t8 = SHL(word1, bptr);
    t9 = SHR(word2, bptr_cmpl);                       /* unsigned shift */
    top1 = t8 + t9;

    Wptr++;
    zzptr++;

/* –––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
 *  Decode AC coefficients
 * ––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––– */
    while (!eob_err)
    {

    /* –––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
     *  Length computation
     * ––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––– */
```

```
/* ----------------------------------------------------------------
 *  _norm returns the number of redundant sign bits in top0, e.g.,
 *  0001 xxxx ... xxxx returns 2, 1111 10xx ... xxxx returns 4.
 *  Example: top=0000 101s xxxx xxxx, then nrm=3.
 * ---------------------------------------------------------------- */
    nrm = _norm(top0);

/* ----------------------------------------------------------------
 *  Eliminate all leading 0s/1s.
 *  Example: t1=0101 sxxx xxxx xxxx
 * ---------------------------------------------------------------- */
    t1  = top0 << nrm;

/* ----------------------------------------------------------------
 *  Use the number of leading bits (norm) as index.
 *  Example: t2= xxxx xxxx 0011 rrrr
 * ---------------------------------------------------------------- */
    t2  = nrm << 4;

    if (intra_vlc_format==0)
        t3 = &len_tbl0[t2];
    else /* intra_vlc_format==1 */
        t3 = &len_tbl1[t2];

/* ----------------------------------------------------------------
 *  Use 4 extra bits after leading bits to distinguish special
 *  cases (32 – 4 = 28).
 * ---------------------------------------------------------------- */
    t4  = t1 >> 28;

/* ----------------------------------------------------------------
 *  For advancing the bitstream, we also need 32-len to shift top1,
 *  and merge it with top0.
 * ---------------------------------------------------------------- */
    if (intra_vlc_format==0)
        t3b = &len_c_tbl0[t2];
    else /* intra_vlc_format==1 */
        t3b = &len_c_tbl1[t2];

    t4b = t1 >> 28;

/* ----------------------------------------------------------------
 *  Get len and 32–len from tables.
 * ---------------------------------------------------------------- */
    len  = t3[t4];
    len_c = t3b[t4b];

/* ----------------------------------------------------------------
 *  Now that we know the length of the current VL code, we can
 *  advance the bitstream to next one:
 *
 *  1. update top0 from top0 and top1.
 * ---------------------------------------------------------------- */
    t5  = top0 << len;
    t7  = top1 >> len_c;
    top0_bk = top0;
    top0 = t5 + t7;
```

```
/* ------------------------------------------------------------------
 *  2. Update top1 from word1 and word2 after increasing bptr by
 *  len. If neccesary (i.e., if new bptr is greater than 32),
 *  update word1 and word2 from memory first. Do not forget that
 *  bptr is always relative to the next lower word boundary, and
 *  therefore needs to be ANDed with 31, in case it become >= 32.
 * ------------------------------------------------------------------- */
    bptr += len;
    test2 = (bptr >= 32);
    if (test2) {
        word1 = word2;
        word2 = bsbuf[next_wptr];
        next_wptr += 1;
        next_wptr &= (BSBUF_SIZE–1);
    }
    bptr = bptr & 31;
    bptr_cmpl = 32 – bptr;
    t8 = SHL(word1, bptr);
    t9 = SHR(word2, bptr_cmpl);                      /* unsigned shift */
    top1 = t8 + t9;

/* -------------------------------------------------------------------
 *  Run-level decode
 * ------------------------------------------------------------------- */

/* -------------------------------------------------------------------
 *  Check if it is an ESCAPE code, which has a unique fixed length
 *  of 24 bits.
 * ------------------------------------------------------------------- */
    test1 = len – 24;

    if (!test1)
    {
    /* ---------------------------------------------------------------
     * ESCAPE code: No look up required, just extract bits: 6 bits
     * for ESCAPE, 6 bits for RUN, then 12 bits for LEVEL.
     * --------------------------------------------------------------- */
        run = _extu(top0_bk, 6, 26);
        level = _ext(top0_bk, 12, 20);
    }
    else
    {
        rld_left = len–5;
        if (len<5) rld_left=0;

        /* -----------------------------------------------------------
         * Last 5 bits of VLC, including sign form, 2nd part of index.
         * ----------------------------------------------------------- */
        rld_index = ((len)<<5) + _extu(top0_bk, rld_left, 32–5);

        if (intra_vlc_format==0)
           run_level = rld_table0[rld_index];
        else /* intra_vlc_format==1 */
           run_level = rld_table1[rld_index];
```

```
            run   = run_level >> 8;
            level = (char)run_level;
        }

        eob_err = (run >= 64);

    /* ------------------------------------------------------------------
     *  Run-lengh expansion and dequantization
     * ------------------------------------------------------------------ */

    /* ------------------------------------------------------------------
     *  Dequantization: *out_i = ((2*level + Sign(level)) * W * qscl)
     * /32. Sign(x)= –1 for x < 0, 0 for x = 0, +1 for x > 0. Division /:
     *  ”Integer division with truncation of the result toward zero.
     *  For example, 7/4 and –7/–4 are truncated to 1 and –7/4, and
     *  7/–4 are truncated to –1.” (MPEG–2 Standard text)
     * ------------------------------------------------------------------ */
        neg = (level < 0);

        f1 = 2*level;

    /* ------------------------------------------------------------------
     *  Find quantization matrix element at zigzag position and
     *  multiply f1 with W * qscl.
     * ------------------------------------------------------------------ */
        if (!eob_err)         /* prevent from accessing memory when EOB */
        {
            W = *(Wptr += run);
            Wptr++;
            /* detect total run overrun */
            eob_err = (Wptr – Wptr_origin)>64;
        }

        qW = qscl * W;
        f3 = f1 * qW;

    /* ------------------------------------------------------------------
     *  For negative numbers, we first need to add 31 before dividing
     *  by 32, to achieve truncation towards zero, as required by the
     *  standard.
     * ------------------------------------------------------------------ */
        if (neg) f3 += 31;

    /* ------------------------------------------------------------------
     *  Saturate to signed 12-bit word (with /32 <–> >>5 incorporated).
     *  SSHL: Shift left, and saturate to 32 bits.
     * ------------------------------------------------------------------ */
        f4 = _sshl(f3, 15);
        f5 = (int)(f4 >> 20);

    /* ------------------------------------------------------------------
     *  Mismatch control: Determine if sum of coefficents is odd or
     *  even.
     * ------------------------------------------------------------------ */
        if (!eob_err)
            sum += f5;
        /* ----------------------------------------------------------------
         *  Find un-zigzag position of DCT coefficient.
         * ---------------------------------------------------------------- */
            if (!eob_err)         /* prevent from accessing memory when EOB */
                cnum = *(zzptr += run);
            zzptr++;
            if (!eob_err)
                outi[block*64+cnum] = f5;

        } /* while */

    /* --------------------------------------------------------------------
     *  Mismatch control: Toggle last bit of last coefficient if sum of
     *  coefficents is even.
     * -------------------------------------------------------------------- */

        if ((sum&1)==0)
        {
            outi[block*64+63] ^= 1;
        }

    /* -------------------------------------------------------------------- */
    /*  Determine nature of fault, invalid code word, or exceeding of the   */
    /*  allowed total run of 64.                                            */
    /* -------------------------------------------------------------------- */
        fault = (run>65) || ((Wptr – Wptr_origin)>64);

        if (fault) break;

    } /* for */

      Mpeg2v[1] = next_wptr;
      Mpeg2v[2] = bptr;
      Mpeg2v[3] = word1;
      Mpeg2v[4] = word2;
      Mpeg2v[5] = top0;
      Mpeg2v[6] = top1;
    Mpeg2v[12]= fault;

}
```

2.6.2 Non-Intra VLD

```
#define BSBUF_SIZE  512 /* bitstream buffer size in words */

#include ”c6000.h”

/* ------------------------------------------------------------------------ */
/*   Run-level tables                                                       */
/* ------------------------------------------------------------------------ */
extern unsigned short rld_table0[576];

/* ------------------------------------------------------------------------ */
/*   Length tables                                                          */
/* ------------------------------------------------------------------------ */
extern const unsigned char len_c_tbl0[512];
extern const unsigned char len_tbl0[512];
```

```
/* ------------------------------------------------------------------------ */
/*  Function                                                                */
/* ------------------------------------------------------------------------ */

void mpeg2_vld_inter_cn(const short *Wptr, short *outi, unsigned int *Mpeg2v)
{

/* ------------------------------------------------------------------------ */
/*  Get bitstream info.                                                     */
/* ------------------------------------------------------------------------ */
    unsigned int        *bsbuf    = (unsigned int *) Mpeg2v[0];
    int                 next_wptr = Mpeg2v[1];
    int                 bptr      = Mpeg2v[2];
    unsigned int        word1     = Mpeg2v[3];
    unsigned int        word2     = Mpeg2v[4];
    unsigned int        top0      = Mpeg2v[5];
    unsigned int        top1      = Mpeg2v[6];
    const unsigned char *zzptr    = (const unsigned char *) Mpeg2v[7];
    int                 qscl      = Mpeg2v[9];
    int                 cbp       = Mpeg2v[11];

    unsigned int bptr_cmpl, top0_bk;

/* ------------------------------------------------------------------------ */
/*  Block number (0–3: lum, 4, 5: chrom)                                    */
/* ------------------------------------------------------------------------ */
    int block;

/* ------------------------------------------------------------------------ */
/*  Variables for length, run and level decoding                            */
/* ------------------------------------------------------------------------ */
    unsigned int t1, t2, t4, t4b, t5, t7, t8, t9;
    int rld_left, rld_index;
    unsigned short run_level;
    unsigned char run, len, len_c;
    const unsigned char *t3, *t3b;
    short level;
    unsigned int eob_err, nrm, fault;
    unsigned int test1, test2;

/* -----------------------------------------------------------------------*/
/*  Variables for dequantization                                           */
/* -----------------------------------------------------------------------*/
    int pos, neg, f1, f3, f5, qW, sum;
#ifdef _TMS320C6X
    int f4;
#else
    _int64 f4;
#endif
    const short *Wptr_origin=Wptr;
    short W;
    unsigned char cnum;

    fault=0;

    for (block=0; block<6; block++)
    {
```

```
/* ------------------------------------------------------------------*/
 *  cbp: Bit 5 4 3 2 1 0. If the corresponding bit is a zero, block no.
 *  (5-bit pos) is not coded.
 * ------------------------------------------------------------------*/
    if (!(cbp & (1 << (5–block))))
        continue;

    zzptr = (const unsigned char *) Mpeg2v[7];
    Wptr=Wptr_origin;

    sum = 0;
    eob_err = 0;

/* ------------------------------------------------------------------
 *  Decode first coefficient.
 *
 *  First code is a special case: When the MSB of the first code is 1,
 *  it can only be VLC 1s (because EOB cannot occur and the
 *  only other code is 11s, which is not valid for the first code).
 *  If MSB = 0, skip to normal AC loop, i.e., use NORM + 4 extra bits as
 *  table index; if MSB=1 execute special case, do not use
 *  NORM because it may have overrun, but set LEN = 2, LEN_C = 30.
 * ------------------------------------------------------------------ */
    if (top0>>31)
    {

    /* --------------------------------------------------------------
     *  Length computation not required, since LEN is 2.
     * -------------------------------------------------------------- */
       len  = 2;
       len_c = 30;

    /* --------------------------------------------------------------
     *  Now that we know the length of the current VL code, we can
     *  advance bitstream to the next one.
     *
     *  1. Update top0 from top0 and top1.
     * -------------------------------------------------------------- */
        t5  = top0 << len;
        t7  = top1 >> len_c;
        top0_bk = top0;
        top0 = t5 + t7;

    /* --------------------------------------------------------------
     *  2. Update top1 from word1 and word2 after increasing bptr by
     *  len. If neccesary (i.e., if new bptr is greater than 32),
     *  update word1 and word2 from memory first. Do not forget that
     *  bptr is always relative to the next lower word boundary, and
     *  therefore needs to be ANDed with 31, in case it become >= 32.
     * -------------------------------------------------------------- */
        bptr += len;
        test2 = (bptr >= 32);
        if (test2) {
            word1 = word2;
            word2 = bsbuf[next_wptr];
            next_wptr += 1;
            next_wptr &= (BSBUF_SIZE–1);
        }
        bptr = bptr & 31;
        bptr_cmpl = 32 – bptr;
        t8 = SHL(word1, bptr);
        t9 = SHR(word2, bptr_cmpl);                   /* unsigned shift */
        top1 = t8 + t9;

    /* ----------------------------------------------------------------
     *  Run-level decode: Run = 0, level = 1.
     * ---------------------------------------------------------------- */
        neg = _extu(top0_bk,1,31);

    /* ----------------------------------------------------------------
     *  Run-lengh expansion and dequantization
     * ---------------------------------------------------------------- */

    /* ----------------------------------------------------------------
     *  Dequantization: *out_i = ((2*level + Sign(level)) * W * qscl)
     * /32. Sign(x)= –1 for x < 0, 0 for x = 0, +1 for x > 0. Division /:
     *  ”Integer division with truncation of the result toward zero.
     *  For example, 7/4 and –7/–4 are truncated to 1, and –7/4 and
     *  7/–4 are truncated to –1.” (MPEG–2 Standard text)
     * ---------------------------------------------------------------- */
        if (neg)
            f1=–3;                                          /* 2*(–1)–1 */
        else
            f1=3;                                              /* 2*1+1 */

    /* ----------------------------------------------------------------
     *  Find quantization matrix element at zigzag position, and
     *  multiply f1 with W * qscl.
     * ---------------------------------------------------------------- */
        W = *Wptr++;
        qW = qscl * W;
        f3 = f1 * qW;

    /* ----------------------------------------------------------------
     *  For negative numbers, we first need to add 31 before dividing
     *  by 32, to achieve truncation towards zero, as required by the
     *  standard.
     * ---------------------------------------------------------------- */
        if (neg) f3 += 31;

    /* ----------------------------------------------------------------
     *  Saturate to signed 12-bit word (with /32 <–> >>5 incorporated).
     *  SSHL: Shift left and saturate to 32 bits.
     * ---------------------------------------------------------------- */
        f4 = _sshl(f3, 15);
        f5 = (int)(f4 >> 20);
        /* ---------------------------------------------------------------
         *  Mismatch control: Determine if sum of coefficents is odd or
         *  even.
         * --------------------------------------------------------------- */
            sum += f5;

        /* ---------------------------------------------------------------
         *  Find un-zigzag position of DCT coefficient.
         * --------------------------------------------------------------- */
            zzptr++;                                         /* always 0 */
            outi[block*64] = f5;
        }

    /* -------------------------------------------------------------------
     *  Decode AC coefficients
     * ------------------------------------------------------------------- */
        while (!eob_err)
        {

        /* ---------------------------------------------------------------
         *  Length computation
         * --------------------------------------------------------------- */

        /* ---------------------------------------------------------------
         *  _norm returns the number of redundant sign bits in top0, e.g.,
         *  0001 xxxx ... xxxx returns 2, 1111 10xx ... xxxx returns 4.
         *  Example: top=0000 101s xxxx xxxx, then nrm=3
         * --------------------------------------------------------------- */
            nrm = _norm(top0);

        /* ---------------------------------------------------------------
         *  Eliminate all the 0s/1s.
         *  Example: t1=0101 sxxx xxxx xxxx
         * --------------------------------------------------------------- */
            t1  = top0 << nrm;

        /* ---------------------------------------------------------------
         *  Use the number of leading bits (norm) as index.
         *  Example: t2= xxxx xxxx 0011 rrrr
         * --------------------------------------------------------------- */
            t2  = nrm << 4;
            t3 = &len_tbl0[t2];

        /* ---------------------------------------------------------------
         *  Use 4 extra bits after leading bits to distinguish special
         *  cases (32–4=28).
         * --------------------------------------------------------------- */
            t4  = t1 >> 28;

        /* ---------------------------------------------------------------
         *  For advancing the bitstream, we also need 32-len to shift top1,
         *  and merge it with top0
         * --------------------------------------------------------------- */
            t3b = &len_c_tbl0[t2];
            t4b = t1 >> 28;
```

```
/* ---------------------------------------------------------------
 *  Get len and 32-len from tables.
 * --------------------------------------------------------------- */
    len  = t3[t4];
    len_c = t3b[t4b];

/* ---------------------------------------------------------------
 *  Now that we know the length of the current VL code, we can
 *  advance the bitstream to the next one:
 *
 *  1. Update top0 from top0 and top1.
 * --------------------------------------------------------------- */
    t5  = top0 << len;
    t7  = top1 >> len_c;
    top0_bk = top0;
    top0 = t5 + t7;

/* --------------------------------------------------------------
 *  2. Update top1 from word1 and word2 after increasing bptr by
 *  len. If neccesary (i.e., if new bptr is greater than 32),
 *  update word1 and word2 from memory first. Do not forget that
 *  bptr is always relative to the next lower word boundary, and
 *  therefore needs to be ANDed with 31, in case it becomes >= 32.
 * --------------------------------------------------------------- */
    bptr += len;
    test2 = (bptr >= 32);
    if (test2) {
        word1 = word2;
        word2 = bsbuf[next_wptr];
        next_wptr += 1;
        next_wptr &= (BSBUF_SIZE–1);
    }
    bptr = bptr & 31;
    bptr_cmpl = 32 – bptr;
    t8 = SHL(word1, bptr);
    t9 = SHR(word2, bptr_cmpl);                    /* unsigned shift */
    top1 = t8 + t9;

/* ---------------------------------------------------------------
 *  Run-level decode
 * --------------------------------------------------------------- */

/* ---------------------------------------------------------------
 *  Check if it is an ESCAPE code, which has a unique fixed length
 *  of 24 bits.
 * --------------------------------------------------------------- */
    test1 = len – 24;

    if (!test1)
    {
        /* ------------------------------------------------------------
         * ESCAPE code: No look-up required, just extract bits: 6 bits
         * for ESCAPE, 6 bits for RUN, then 12 bits for LEVEL.
         * ------------------------------------------------------------ */
            run = _extu(top0_bk, 6, 26);
            level = _ext(top0_bk, 12, 20);
        }
        else
        {
            rld_left = len–5;
            if (len<5) rld_left=0;

            /* --------------------------------------------------------
             *  Last 5 bits of VLC include sign form, 2nd part of index.
             * -------------------------------------------------------- */
            rld_index = ((len)<<5) + _extu(top0_bk, rld_left, 32–5);
            run_level = rld_table0[rld_index];

            run   = run_level >> 8;
            level = (char)run_level;
        }

        eob_err = (run >= 64);

    /* ----------------------------------------------------------------
     *  Run-length expansion and dequantization
     * ---------------------------------------------------------------- */

    /* ----------------------------------------------------------------
     *  Dequantization: *out_i = ((2*level + Sign(level)) * W * qscl)
     * /32. Sign(x)= –1 for x < 0, 0 for x = 0, +1 for x > 0. Division /:
     *  ”Integer division with truncation of the result toward zero.
     *  For example, 7/4 and –7/–4 are truncated to 1, and –7/4 and
     *  7/–4 are truncated to –1.” (MPEG–2 Standard text)
     * ---------------------------------------------------------------- */
        neg = (level < 0);

        f1 = 2*level;

     /* ----------------------------------------------------------------
      *  This needs to be added over intra.
      *  --------------------------------------------------------------- */
        pos = (level > 0);
        f1 = f1 – neg;
        f1 = f1 + pos;
        /* ----------------------------------------------------------------
         *  Find quantization matrix element at zigzag position, and
         *  multiply f1 with W * qscl.
         * ---------------------------------------------------------------- */
            if (!eob_err)         /* prevent from accessing memory when EOB */
            {
                W = *(Wptr += run);
                Wptr++;
                /* detect total run overrun */
                eob_err = (Wptr – Wptr_origin)>64;
            }

            qW = qscl * W;
            f3 = f1 * qW;

        /* ----------------------------------------------------------------
         *  For negative numbers, we first need to add 31 before dividing
         *  by 32, to achieve truncation towards zero, as required by the
         *  standard.
         * ---------------------------------------------------------------- */
            if (neg) f3 += 31;

        /* ----------------------------------------------------------------
         *  Saturate to signed 12-bit word (with /32 <–> >>5 incorporated).
         *  SSHL: Shift left and saturate to 32 bits.
         * ---------------------------------------------------------------- */
            f4 = _sshl(f3, 15);
            f5 = (int)(f4 >> 20);

        /* ----------------------------------------------------------------
         *  Mismatch control: Determine if sum of coefficents is odd or
         *  even.
         * ---------------------------------------------------------------- */
            if (!eob_err)
                sum += f5;

        /* ----------------------------------------------------------------
         *  Find un-zigzag position of DCT coefficient.
         * ---------------------------------------------------------------- */
            if (!eob_err)         /* prevent from accessing memory when EOB */
                cnum = *(zzptr += run);
            zzptr++;
            if (!eob_err)
                outi[block*64+cnum] = f5;

        } /* while */

    /* --------------------------------------------------------------------
     *  Mismatch control: Toggle last bit of last coefficient if sum of
     *  coefficents is even.
     * -------------------------------------------------------------------- */

        if ((sum&1)==0)
        {
            outi[block*64+63] ^= 1;
        }
        /* --------------------------------------------------------------- */
        /*  Determine nature of fault, invalid code word, or exceeding of the   */
        /*  allowed total run of 64.                                           */
        /* --------------------------------------------------------------- */
            fault = (run>65) || ((Wptr – Wptr_origin)>64);

            if (fault) break;

        } /* for */

        Mpeg2v[1] = next_wptr;
        Mpeg2v[2] = bptr;
        Mpeg2v[3] = word1;
        Mpeg2v[4] = word2;
        Mpeg2v[5] = top0;
        Mpeg2v[6] = top1;
        Mpeg2v[12]= fault;
    }
```

2.7 Optimizations for C62x

To achieve the best possible performance, it is recommended to code the VLD routines in hand-optimized assembly. Presently, the assembly optimizer will not be able to achieve as good as a performance compared to hand-optimized assembly code. The main reason is the irregular trip count of the AC decoding loop. Irregular trip count means that the trip count is not known at the time the loop is entered (“while” loop). The loop may still be software pipelined, but the handling of live-out variables is more complicated because it is not known after which iteration the loop exits. Live-out variables are variables whose values are required after the loop exits, such as, for instance, all the variables that describe the current state of the bitstream: top0, top1, bptr, etc. They are required either for decoding the next block or for updating the bitstream context structure Mpeg2v before leaving the routine. When it is known that the loop exits (i.e., eob_err flag is set) live-out variables may already have been overwritten by instructions of extra iterations, which should not have been executed, but were already initiated. The assembly optimizer tries to preserve variables for live-out by predicating instructions that would modify those variables. In case of the VLD routines, the number of available predicate registers is insufficient due to the large number of live-out variables. This prevents the assembly optimizer from software pipelining the loop. The issue of live-out registers is also important for the hand code, and is discussed in more detail in section 2.7.6.

2.7.1 Critical Recurrence Path

The initiation interval of the AC coefficients decoding loop is bound by a recurrence path inherent to the variable-length decoding problem. Before the next code word can be decoded the length of the current code must first be decoded, and the bit stream advanced accordingly, i.e., the next iteration can only be initiated after top0 has been updated. The assembly code for computing the code length and updating top0 may be written as:

```
1            NORM.L1 A_top0,        A_nrm

2            SHL     A_top0, A_nrm,  A_t1                ; Eliminate leading 0s/1s.
         ||  SHL     A_nrm,  4,      B_t2                ; Determine sub-table index.

3            ADD     B_len_tbl_adr,  B_t2,   B_t3
         ||  SHRU    A_t1,     28,   B_t4                ; 4 extra bits after leading bits

4            LDBU    *B_t3[B_t4],    A_len               ; Get length.

5,6,7,8      NOP 4

9            SUB     const32, len,   B_len_c             ; Get 32-length.

10           SHL     A_top0,         A_len,        A_t6
         ||  SHRU    B_top1,         B_len_c,      B_t7

11           ADD     A_t6,           B_t7,         A_top0 ; Update top0
```

Table 6 shows schematically how multiple iterations can overlap. Before the NORM instruction of the following iteration is issued, top0 of the previous iteration must have been updated. Thus, the recurrence bound is RecMII = 11 for the above schedule.

Table 6. Overlapping Multiple Iterations of the AC Decoding Loop

First Iteration	Second Iteration	Third Iteration
NORM A_top0		
...		
ADD A_t6, B_t7,A_ top0		
(Update top1, bptr, word1, word2.)	NORM A_top0	
(Run-level decode)	...	
(Dequantization)	ADD A_t6, B_t7,A_ top0	
(Store result.)	(Update top1, bptr, word1, word2)	NORM A_top0
	(Run-level decode)	...
	(Dequantization)	ADD A_t6, B_t7,A_ top0
	(Store resul.t)	(Update top1, bptr, word1, word2.)
		(Run-level decode)
		(Dequantization)
		(Store result.)

As already discussed previously and realized in the C model, the recurrence path can be shortened by using a look-up table to obtain the value (32-len) instead of calculating it. The instructions on the recurrence path then are:

```
1             NORM    A_top0,         A_nrm

2             SHL     A_top0, A_nrm,  A_t1                ; Eliminate all leading 0s/1s.

          ||  SHL     A_nrm,  4,      B_t2                ; Determine sub-table index.

3             ADD     B_len_tbl_adr,  B_t2,   B_t3
          ||  ADD     A_len_c_tbl_adr,B_t2,   A_t3b
          ||  SHRU    A_t1,     28,   B_t4                ; 4 extra bits after leading bits
          ||  SHRU    A_t1,     28,   A_t4b               ; 4 extra bits after leading bits

4             LDBU    *B_t3[B_t4],    A_len               ; Get length.
          ||  LDBU    *A_t3b[A_t4b],  B_len_c             ; Get 32-length.

5,6,7,8       NOP 4

9             SHL     A_top0,      A_len,        A_t6
          ||  SHRU    B_top1,      B_len_c,      B_t7

10            ADD     A_t6,        B_t7,         A_top0    ; Update top0.
```

Now the recurrence path has been shortened by one cycle to RecMII=10. Figure 4 illustrates the new recurrence path in a graph-style diagram.

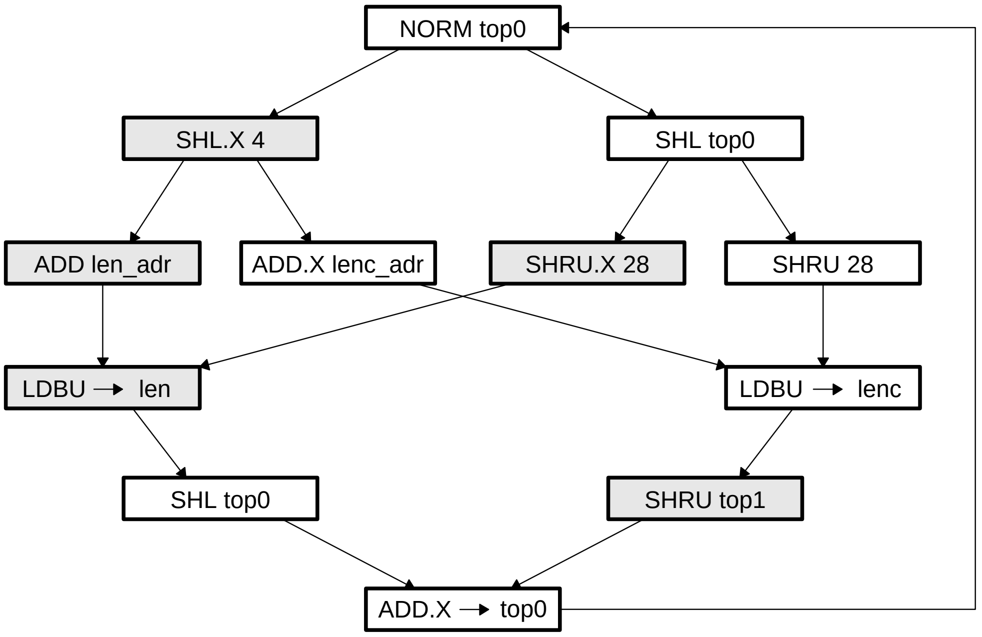


Figure 4. Critical Recurrence Path

2.7.2 *Circular-Addressing Mode*

In assembly code, we can make use of the circular-addressing mode which is available for some registers. This allows us to simplify the update of the bitstream register, word2, from the bitstream buffer. In C the following code was necessary:

```
                  word2 = bsbuf[next_wptr];
                  next_wptr += 1;
                  next_wptr &= buf_mask;
```

Using the circular-addressing mode, this reduces to:

```
                  LDW     *bsbuf_circ++,   word2
```

The register bsbuf_circ has to be setup as circular in the AMR control register. Assuming the buffer size is 512 words and the register to be used is B7, the AMR register has to be set up with:

```
     MVKL    0x00AA4800, B0  ; B7 circular mode, 2^(10+1) bytes
     MVKH    0x00AA4800, B0  ; (= 2048 bytes = 512 words x 32 bit)
     MVC     B0, AMR
```

2.7.3 General Optimizations

From the functional-unit resources point of view, the .S unit tends to be the major bottleneck due to the high number of shift operations required. It may be necessary in some cases to find alternative instructions in order to balance the functional unit usage better. For instance, *SUB* can be used instead of CMPEQ, MPY 0,x,y instead of ZERO y, or MPY x,32,y instead of SHL x,5,y. Also, instead of extracting the sign bit from the VLC code, which would require an EXT instruction, the method of looking up the signed level was chosen as described earlier.

2.7.4 Partitioned Linear Assembly Code for AC Decoding Loop

Here the partitioned linear assembly code for the AC decoding loop is shown. The code follows the C model and integrates the optimizations discussed above. AC decoding for intra and non-intra is very similar, with only two instructions more required for dequantization in the non-intra case. In the following assembly code, these two instructions are marked with “for non-intra only”. The meaning of “preserve” in the comments of the code below is explained in section 2.7.5.

```
loop:
* ----------------------------------------------------------------
*  Length computation
* ---------------------------------------------------------------- *
            NORM    A_top0,         A_nrm
            SHL     A_top0, A_nrm,  A_t1          ; Eliminate all leading 0s/1s.
            SHL     A_nrm,  4,      B_t2          ; Determine sub-table index

            ADD     B_len_tbl_adr,  B_t2,   B_t3
            ADD     A_len_c_tbl_adr,B_t2,   A_t3b

            SHRU    A_t1,     28,   B_t4          ; 4 extra bits after leading bits
            SHRU    A_t1,     28,   A_t4b         ; 4 extra bits after leading bits

            LDBU    *B_t3[B_t4],    A_len         ; Get length.
            LDBU    *A_t3b[A_t4b],  B_len_c       ; Get 32-length.
```

```
* --------------------------------------------------------------------
*  Advance bitstream (assumes little endian)
* -------------------------------------------------------------------- *
            SHL     A_top0,      A_len,        A_t6
            SHRU    B_top1,      B_len_c,      B_t7
            MV      A_top0,                    B_top0_bk
[!B_eob_err]ADD     A_t6,        B_t7,         A_top0   ; Preserve top0.

            ADD     A_bptr,      A_len,        A_bptr1
            CMPGT   A_bptr1,     A_const31,    A_test2
[B_eob_err] MPY     0,           A_const31,    A_test2  ; Or ZERO A_test2

[A_test2]   MV      B_word2,                   A_word1  ; Preserve word1
[A_test2]   LDW     *B_bsbuf_circ++,          B_word2   ; Preserve bsbuf_circ, word2

[!B_eob_err]AND     A_const31,   A_bptr1,      A_bptr   ; Preserve bptr
            SUB     B_const32,   A_bptr,       B_bptr_cmpl

            SHL     A_word1,     A_bptr,       A_t8
            SHRU    B_word2,     B_bptr_cmpl,  B_t9
[!B_eob_err]ADD     A_t8,        B_t9,         B_top1   ; Preserve top1

* ----------------------------------------------------------------
*  Run-level escape code decode
* ---------------------------------------------------------------- *
            SUB     A_len,        24,     A_test1
            EXTU    B_top0_bk,    6,      26,     B_run
            EXT     B_top0_bk,    12,     20,     B_level

* ----------------------------------------------------------------
*  Run-level “normal” code decode
* ---------------------------------------------------------------- *
            CMPLT   A_len,        5,          B_test3    ; If len<=4. do not shift. left
            SUB     A_len,        5,          B_rld_left
[!B_test3]  MPY     B_const32,    0,          B_rld_left ; Or ZERO B_rld_left.

            MPY     B_const32,    A_len,        A_t12    ; Or SHL A_len, 5, A_t12.
            SHL     B_top0_bk,    B_rld_left,   B_t13    ; Align last 5 bits of VLC at MSB.
            SHRU    B_t13,        27,           B_t14    ; Keep last 5 bits (including sign).
            ADD     B_t14,        A_t12,        A_t15    ; Add to sub-table index.
            ADD     A_t15,        A_t15,        A_t16    ; Half-word entries
[A_test1]   LDB     *A_rld_table_adr[A_t16],    B_level  ; Access
[A_test1]   LDBU    *A_rld_table_adr_1[A_t16],  B_run    ; A_rld_table_adr+1
```

```
* ----------------------------------------------------------------- *

            CMPGT   B_run,    B_const63,         B_eob_err
            MV      B_run,    B_run_bk           ; Preserve run for error. classification

* -----------------------------------------------------------------
*  De-quantization
* ----------------------------------------------------------------- *

            CMPLT   B_level,    0,          A_neg

            MPY     B_level,    2,          B_level3

; for non-intra only:
;[ A_neg ]   SUB     B_level3,   1,          B_level3
;[!A_neg ]   ADD     B_level3,   1,          B_level3

            LDH     *++B_Wptr[B_run],       A_W
[!B_eob_err]CMPGT   B_Wptr,     B_Wptr_end, B_eob_err   ; Error detect
            ADD     B_Wptr,     2,          B_Wptr
            MPY     A_qscl,     A_W,        A_qw
            MPY     A_qw,       B_level3,   A_level4
[A_neg]     ADD     A_level4,   A_const31,  A_level4

            SSHL    A_level4,   15,         A_level5
            SHR     A_level5,   20,         A_level_f

            ADD     A_sum,      A_level_f,  A_sum

            LDB     *++B_Zptr[B_run],       A_Z
            ADD     B_Zptr,    1,           B_Zptr

[!B_eob_err]STH     A_level_f,  *A_outi++[A_Z]

[!B_eob_err]B       loop
```

2.7.5 Live-Out Variables

When the loop exits, the correct state of some registers needs to be preserved. These are the bitstream management variables (top0, top1, bptr, word1, word2, bsbuf_circ), sum and run for error classification. In the partitioned linear assembly code shown in section 2.7.4, instructions that could modify the content of these registers are predicated so that their state at the time eob_err was set is preserved. The variable run is moved into another register that can be made live-out. Apart from predicating instructions and moving variables to save the contents of registers, variables could be preserved by putting them on the stack, reconstructing, or making their registers live throughout the whole loop (i.e., they do not share a register with any other variables). The above code may actually be too restrictive for the C6000 architectures, due to high register pressure. For instance, the instruction modifying top1 is predicated, but this does not mean that the register cannot be written by another instruction. If the predicated instruction is scheduled further away from the loop exit, it is more likely that the same register is still used by other instructions. For the predicate to be effective, top1 has to be assigned an appropriate register which will not be overwritten before the loop exits. Therefore, the methods that should be used for variable preservation depend on the actual software pipelined schedule. It may be helpful to software-pipeline the loop with fewer preservation predicates and instructions, and fit them into the scheduled loop afterwards. Note that when the loop is pipelined, i.e., multiple iterations are overlapped, instructions that appear before eob_err is defined may then appear as instructions of the following iterations after the definition of eob_err of the previous iteration. Therefore, these instructions have to be predicated. Depending on the pipelined schedule, eob_err may be set only after the loop branch has already occurred. In this case, instructions have to be predicated that appear before the eob_err definition in the pipelined loop because the kernel executes one more time.

2.7.6 Error Behavior Analysis

In this section we will examine the validity of memory accesses, in case invalid code words are encountered.

No invalid memory access can occur when updating the bitstream registers from the bitstream buffer in memory, since it is circularly addressed through B_bsbuf_circ.

Table indices for the length look-up can lie in the range 0...504. The largest index of 504 occurs when all 32 bits of top0 are set to 1. Then, norm returns 31, and the 4 extra bits are 1000b (remember that the extra bits are the 4 most significant bits of top0<<norm), i.e., 31 << 4 + 1000b = 504. The length look-up tables have been implemented with 512 entries, and thus cover this maximum range (the seven extra entries at the end are never accessed, and could be removed if necessary). Entries of invalid table indices are set to the invalid length of zero so that the eob_err flag will be set when a subsequent run-level look-up is performed. Remember that the length is used as sub-table index for the run-level look-up, and that all invalid run-level table entries return a run value of 65 (64 is used to signal EOB), which causes the eob_err flag to be set.

To examine memory accesses into the run-level look-up tables, we have to look at the code length looked-up previously, since it is now used as a sub-table index (i.e., length + 5 right-most bits of the code word forms the index into the run-level look-up table). Note that the code length is guaranteed to be either 0 (if an invalid code occurred) or lie in the range 3...17 for Table 2, and 2...17 for Table B–14. Escape sequences have a length of 24, but they do not need to be considered here, since their run-level values are directly extracted from the code word and not looked up in the table. Since we have ensureded bounds, there never occurs an invalid memory access if the tables cover the indices 0 to 575 (= 17<<5 + 11111b).

Note that since the flag eob_err will be set when run > 63, it cannot be determined at first whether a regular EOB code (run entry 64) was decoded or an error (run entry 65) occurred. Since it may be a valid block, it is important to ensure that all live-outs and the result array are preserved (and not altered by operations of the following iterations which are already in the pipeline) when the decoding loop exits. In the outer loop, we can then detect if it was an error by making the variable run live-out and checking if run > 64. If it was an error, the outer loop should break and return a set fault flag to the caller, which can then initiate resynchronization.

In some cases a valid VLC code may be decoded, but the run value causes the total run to exceed 63, which may consequently result in an illegal write memory access beyond the end of the output array. It also results in a read memory access beyond the weighting matrix and inverse zigzag matrix arrays. To prevent the write access, we can use the pointer of, for instance, the weighting matrix Wptr and compare it to the end address of the array, Wptr_end. If, after Wptr has been updated with the current run value, Wptr is greater than Wptr_end, then the total run exceeded 63, and the eob_err is set:

```
CMPGT   B_Wptr,     B_Wptr_end, B_eob_err
```

Assuming that the store instruction is predicated on eob_err and that the compare instruction can be issued before the store occurs, no illegal write access will occur.

In other cases, the compare instruction cannot be issued before the store instruction, because no instruction slot is available due to the structure of the software pipelined loop. An illegal write memory access can still be prevented in the current iteration by addressing the output array in circular mode. For the next iteration the eob_err flag will be set, and the loop exits.

In the outer loop, it should then be checked if Wptr > Wptr_end, additionally to run > 64, to distinguish between an error condition and an EOB code.

Since the pointer, Wptr, is allowed to go passed the end of the array, once in order to be able detect a total run overrun, the array should be padded appropriately. The maximum overrun that can occur is 66 which is the run value chosen as error mark. Therefore, the weighting matrix array should be padded at the end with 66 + 1 half-words.

2.7.7 Software Pipelined AC Decoding Loop (Intra)

Here the hand-optimized kernel of the pipelined AC decoding loop is shown for the intra case. Each instruction is annotated with the number of the instruction in the flat (non-pipelined) schedule and the number of the iteration it belongs to. As discussed above, instructions have been added to preserve the correct state of the variables required to be live-out. The initiation interval is 10 cycles, bound by the recurrence path for updating top0. The resource bound is ResMII = 8 (without the preserving instructions).

```
* ========================== PIPE LOOP KERNEL ============================ *
loop:
             ADD    .L1     A_t15,      A_t15,      A_t16          ;[17,2]
||           SUB    .D1     A_len,      24,         A_test1        ;[17,2]
||           SHRU   .S2     B_word2,    B_bptr_cmpl,B_t9           ;[17,2]
||           SHL    .S1     A_word1,    A_bptr,     A_t8           ;[17,2]
||           STW    .D2T2   B7_bsbuf_circ, *+B_SP[BSBUFC_SPOFF]     ;preserve

             MPY    .M1     A15_qscl,   A_W,        A_qw           ;[28,1]
||[A_test1]  LDBU   .D1T2   *A_rld_table_adr_1[A_t16],    B_run    ;[18,2]
||           EXTU   .S2     B8_top0_bk,  6,  26,    B_run          ;[18,2]
||[!B_eob_err]ADD   .L2X    A_t8,       B_t9,       B8_top1        ;[18,2]
||[B_eob_err] MV    .D2     B_run,      B5_run_bk                  ;preserve
||[B_eob_err] B     .S1     exit                                   ;EOB_ERR exit
```

```
  [A_test1]    LDB   .D1T2    *A_rld_table_adr[A_t16],B_level          ;[19,2]
||             CMPLT .L2X     A_len,       5,          B_test3          ;[ 9,3]
||             ADD   .L1      A_bptr,      A_len,      A_bptr1          ;[ 9,3]
||             SHRU  .S2      B8_top1,     B_len_c,    B_t7             ;[ 9,3]
||             SHL   .S1      A_top0,      A_len,      A_t6             ;[ 9,3]

  [!B_eob_err]CMPGT .L1      A_bptr1,     A14_const31,A_test2          ;[10,3]
||[B_eob_err] MPY   .M1      0,           A14_const31,A_test2          ;preserve
||             ADD   .S1X     A_t6,        B_t7,       A_top0           ;[10,3]
||[!B_eob_err]MV    .S2X     A_top0,      B8_top0_bk                   ;[10,3]prsrv
||[!B_eob_err]STW   .D2T1    A_bptr,      *+B_SP[BPTR_SPOFF]           ;preserve
||[!B_eob_err]ADD   .L2      B9_Wptr,     2,          B9_Wptr          ;[31,1]prsrv

  [!B_eob_err]B     .S2      loop                                      ;[31,1]
||             MPY   .M1X     A_qw,        B_level3,   A_level4         ;[31,1]
||[B_test3]    MPY   .M2      B_const32,   0,          B_rld_left       ;[11,3]
||             SUB   .L2X     A_len,       5,          B_rld_left       ;[11,3]
||             AND   .S1      A14_const31,A_bptr1,     A_bptr           ;[11,3]
||             NORM  .L1      A_top0,      A_nrm                        ;[ 1,4]
||[!B_eob_err]ADDAB .D2      B5_Zptr,     1,          B5_Zptr          ;[27,1]

  [ A_test2]   LDW   .D2T2    *B7_bsbuf_circ++,        B_word2          ;[12,3]
||[ A_test2]   MV    .L1X     B_word2,     A_word1                      ;[12,3]
||             SHL   .S2X     A_nrm,       4,          B_t2             ;[ 2,4]
||             SHL   .S1      A_top0,      A_nrm,      A_t1             ;[ 2,4]

  [ A_neg]     ADD   .D1      A_level4,    A14_const31,A_level4         ;[33,1]
||[!B_eob_err]LDH   .D2T1    *++B9_Wptr[B_run],       A_W              ;[23,2]prsrv
||             SHRU  .S1      A_t1,        28,         A_t4b            ;[ 3,4]
||             SHRU  .S2X     A_t1,        28,         B_t4             ;[ 3,4]
||             ADD   .L1X     A_len_c_tbl_adr,         B_t2,     A_t3b ;[ 3,4]
||             ADD   .L2      B_len_tbl_adr,           B_t2,     B_t3  ;[ 3,4]
; EOB_ERR branch occurs

               SSHL  .S1      A_level4,    15,         A_level5         ;[34,1]
||             SHL   .S2      B8_top0_bk,  B_rld_left, B_t13            ;[14,3]
||             MPY   .M1X     B_const32,   A_len,      A_t12            ;[14,3]
||             SUB   .L2X     B_const32,   A_bptr,     B_bptr_cmpl      ;[14,3]
||             LDBU  .D1T2    *A_t3b[A_t4b],           B_len_c          ;[ 4,4]
||             LDBU  .D2T1    *B_t3[B_t4],             A_len            ;[ 4,4]

               SHR   .S1      A_level5,    20,         A3_level_f       ;[35,1]
||             LDB   .D2T1    *++B5_Zptr[B_run],       A_Z              ;[25,2]
||          MPY   .M2     B_level,    2,          B_level3        ;[25,2]
||          CMPLT .L1X    B_level,    0,          A_neg           ;[25,2]
||          SHRU  .S2     B_t13,      27,         B_t14           ;[15,3]
||          CMPGT .L2     B_run,      B14_const63,B_eob_err       ;[26,2]

            STH   .D1T1   A3_level_f, *+A_outi[A_Z]               ;[36,1]
||          ADD   .S1     A_sum,      A3_level_f, A_sum           ;[36,1]
||          ADD   .L1X    B_t14,      A_t12,      A_t15           ;[16,3]
||          EXT   .S2     B8_top0_bk, 12, 20,     B_level         ;[16,3]
||[B_eob_err] MPY   .M2     0,          B14_const63,B9_Wptr         ;Rule out overrun.
||[!B_eob_err]CMPGTU.L2     B9_Wptr,    B0_Wptr_end,B_eob_err       ;err detec

* ========================== PIPE LOOP EPILOG ============================ *
```

The epilog can be completely removed, and the prolog partially overlapped with intra DC decoding. For the non-intra case, the software pipelined loop will look very similar. In this case, the prolog can be modified to account for the special case of the first coefficient.

2.7.8 Implementation Notes and Special Requirements

Note that this is a little endian implementation, and that it is assumed that the bitstream data is stored in memory in 32-bit elements, with little endian byte ordering. All routines are non-interruptible for their entire duration.

The bitstream buffer is set to 512 32-bit words (= 2048 bytes) and needs to be aligned at a 2048 boundary because it is circularly addressed. If the size needs to be changed, the AMR register setup has to be modified, and the alignment changed accordingly.

The zigzag matrix (Zptr) is 64 bytes in size, circularly addressed, and consequently needs to be aligned at a 64-byte boundary. Circular addressing of the matrix serves as protection from illegal stores into memory if invalid VLC codes occur.

The pointer, Wptr, is allowed to go past the end of the array once, to be able detect a total run overrun. The maximum overrun that can occur is 66 which is the run value chosen as an error mark. Therefore, the weighting matrix array should be padded at the end with 66 + 1 half-words. No memory is overwritten, only loads occur.

Inside the routines, word1 (next_wptr-2) and word2 (next_wptr-1) are reconstructed from the bitstream buffer, and therefore have to be kept alive in the bitstream buffer. For instance, in a double-buffering scheme, the bitstream buffer can only be updated when next_wptr-2 (and not next_wptr) has crossed the half-buffer boundary.

Also, note that the layout of the Mpeg2v bitstream context structure should be kept fixed, since the variables are being accessed by the assembly routine with absolute references. If the layout is changed, the corresponding changes have to be made in the assembly code too.

2.8 Optimizations for C64x

As opposed to the C62x, the C64x inserts delay slots when a cross path is used to read a result written in the previous cycle. Therefore, the length computation presented above for the C62x having 10 cycles would now require 13 cycles on the C64x due to cross path delay slots. One way to avoid cross path delay slots is to schedule all instructions on one side. For instance:

```
1            NORM.L1 A_top0,         A_nrm
2            SHL .S1 A_top0, A_nrm,  A_t1               ; Eliminate all leading 0s/1s.
3            SHL .S1 A_nrm,  4,      A_t2               ; Determine sub-table index.
4            ADD     A_len_tbl_adr,  A_t2,  A_t3
          || SHRU    A_t1,     28,   A_t4               ; 4 extra bits after leading bits
5            LDBU    *A_t3[A_t4],    A_len              ; Get length.
6,7,8,9      NOP 4
10           SHL     A_top0,      A_len,    A_t6
          || SUB     A_const32,   A_len,    A_len_c       ; Get 32-length.

11           SHRU    A_top1,      A_len_c,  A_t7
12           ADD     A_t6,        A_t7,     A_top0        ; Update top0.
```

This reduces the critical recurrence path to 12 cycles, but is still 2 cycles more, compared to the C62x version. The reason why we need to wait until top0 is fully updated from top1 is that the next VLC code may have more bits than there are remaining bits in top0. However, if we can ensure that after advancing top0 to the start of the next code with the instruction SHL A_top0, A_len, A_t6, this next VLC code is fully contained in top0, and we could initiate a new iteration beginning with NORM right after this instruction. The remaining instructions required for a complete update, which are the SUB, SHRU and ADD instructions in execute packets 10 to 12, can then be executed in parallel with, or after, NORM. Note that it is only required that sufficient bits be available, so that NORM can operate correctly. For instance, in the case where the next code is an escape sequence, only the VLC part of the sequence, which is 6 bits, needs to be available. By the time run and level is decoded, top0 will have been fully updated.

Assuming the longest possible code sequence has occurred and the bitstream has been advanced by its length, then top0 must contain at least the number of bits equal to the longest VLC code in order for NORM to work correctly (note that NORM does not rely on the sign bit of the VLC). The longest code length occurring in MPEG-2 is the escape sequence with 24 bits: 6 bits for VLC code, 6 bits for run, and 12 bits for level. The longest VLC code, without counting the sign bit, is 16 bits long. Therefore, the worst case is that an escape sequence has occurred, and the following VLC code is 16 bits. This requires that top0 contains at least 24 + 16 = 40 bits. This can be achieved by using a 40-bit register pair for top0, A_top0h:A_top0l:

```
1            NORM.L1 A_top0h:A_top0l,  A_nrm
2            SHL .S1 A_top0h:A_top0l,  A_nrm,  A_t1h:A_t1l    ; Eliminate all leading 0s/1s.
3            SHL .S1 A_nrm,  4,        A_t2                   ; Determine sub-table index.
4            ADD     A_len_tbl_adr,    A_t2,      A_t3
          || SHRU    A_t1h:A_t1l,      A_const36, A_t4h:A_t4l ; 4 extra bits
5            LDBU    *A_t3[A_t4l],     A_len                  ; Get length.
6,7,8,9      NOP 4
10           SHL     A_top0h:A_top0l,  A_len,     A_top0h:A_top0l
```

This reduced the recurrence path to 10 cycles. Figure 5 illustrates the new 40-bit/32-bit register-based bitstream management.

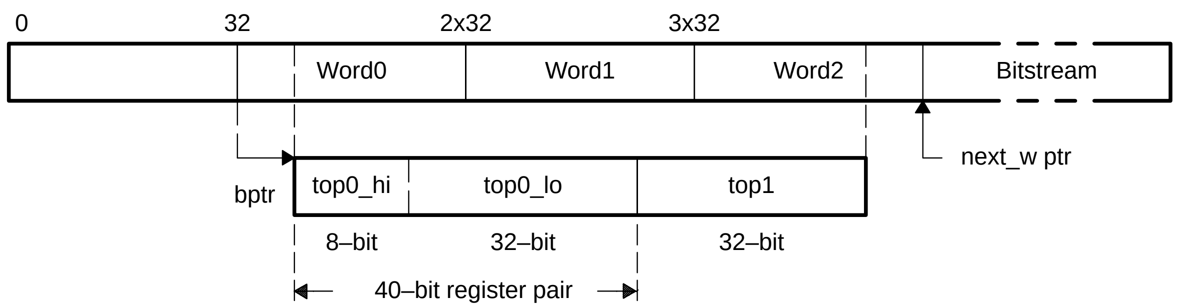


Figure 5. Bitstream Management for C64x

To complete the update, we further have to perform:

```
SUB     A_const32,        A_len,     A_len_c
SHRU    A_top1,           A_len_c,   A_t7
ADD     A_top0l,          A_t7,      A_top0l
```

Now A_top0h:A_top0l contains the fully updated bitstream. However, because we use the same register names for both the partial update in instruction 10 and the full update here, we actually tie in the full update in the recurrence path.To avoid this, rename the *destination* register, A_top0h:A_top01, in instruction 10, and the source register, A_top0h:A_top01, in instructions 1 and 2, to A_ptop0h:A_ptop0l.

The actual run-level decoding part and dequantization remain the same as for the C62x; however, the VLC length look-up tables need to be extended to reflect the fact that NORM now operates on a 40-bit register pair, instead of a 32-bit register. For the C62x the largest index was determined by the case that all 32 bits of top0 are 1, and norm returns 31. In the case of C64x, 40 bits may be 1, and norm returns 39. Now 6 bits are required to represent norm, and with the 4 extra bits, we require a 10-bit index for the length look-up tables. However, since the maximum is 39<<4 + 1000b, we only require 8 x 16 = 128 entries more compared to the C62x version. These extra entries need to contain the length of the code words, 11s for Table B–14, and 11111111s for Table 2 (see section 2.4.1).

Here is the loop serial assembly code:

```
loop:
* ----------------------------------------------------------------
*  Length computation
* ---------------------------------------------------------------- *
            NORM    A_ptop0h:A_ptop0l,   A_nrm                  ; hi8:lo32

            SHL     A_ptop0h:A_ptop0l,   A_nrm,     A_t1h:A_t1l

            SHL     A_nrm, 4,         A_t2

            ADD     A_len_tbl_adr,    A_t2, A_t3
            SHRU    A_t1h:A_t1l,      A_const36, A_t4h:A_t4l    ; 4 extra bits

            LDBU    *A_t3[A_t4l],     A_len                     ; Get length.

            SUB     A_const32,        A_len,     A_len_c
```

```
* -------------------------------------------------------------------
*  Advance bitstream.
* ------------------------------------------------------------------- *
            SHL     A_top0h:A_top0l,  A_len,   A_ptop0h:A_ptop0l

            SHRU    A_top1,           A_len_c, A_t7
            SHRU    A_top0h:A_top0l,  8,       A_empty:A_top0_bk

            MV      A_ptop0h,                  A_top0h
            ADD     A_ptop0l,    A_t7,         A_top0l

            ADD     A_bptr,      A_len,        A_bptr1
            CMPGT   A_bptr1,     A_const31,    A_test2

 [A_test2]   MV      B_word2,                   A_word1
 [A_test2]   LDW     *B_bsbuf_circ++,           B_word2

            AND     A_const31,   A_bptr1,      A_bptr
            SUB     B_const32,   A_bptr,       B_bptr_cmpl

            SHL     A_word1,      A_bptr,      A_t8
            SHRU    B_word2,      B_bptr_cmpl, B_t9
            ADD     A_t8,         B_t9,        A_top1

* ---------------------------------------------------------------
*  Run–Level escape code decode
* --------------------------------------------------------------- *
            SUB     A_len,        A_const24,   A_test1       ; Check if escape code.
            MV      A_top0_bk,    B_top0_bk
            EXTU    B_top0_bk,    6,      26,  B_run
            EXT     B_top0_bk,    12,     20,  B_level

* ---------------------------------------------------------------
*  Run-level ”normal” code decode
* --------------------------------------------------------------- *
            CMPLT   A_len,        5,        B_test3   ; If len <=4, no left-shift.
            SUB     A_len,        5,        B_rld_left
[!B_test3]  MPY     B_rld_left,   0,        B_rld_left

            MPY     B_const32,    A_len,        B_t12  ; <<5
            SHL     B_top0_bk,    B_rld_left,   B_t13  ; Align at MSB.
            SHRU    B_t13,        27,           B_t14  ; Last 5 bits(including sign)
            ADD     B_t14,        B_t12,        B_t15  ; Add to sub-table index.
            ADD     B_t15,        B_t15,        B_t16  ; Table is half-word.

[A_test1]   LDB     *B_rld_table_adr[B_t16],    B_level
[A_test1]   LDB     *B_rld_table_adr_1[B_t16],  B_run  ; A_rld_table_adr+1

* --------------------------------------------------------------- *

            CMPGT   B_run,    B_const63,        B_eob_err

* ---------------------------------------------------------------
*  Dequantization
* --------------------------------------------------------------- *

            CMPLT   B_level,    0,          A_neg

            MPY     B_level,    2,          B_level2
```

```
; for non-intra only:
;            SUB     B_level2,   1,         B_level3
;[!A_neg ]   ADD     B_level2,   1,         B_level3

            LDB     *++B_Wptr[B_run],       A_W
            ADD     B_Wptr,     1,          B_Wptr

            MPY     A_qscl,     A_W,        A_qw
            MPY     A_qw,       B_level3,   A_level4
[A_neg]     ADD     A_level4,   A_const31,  A_level4

            SSHL    A_level4,   15,         A_level5
            SHR     A_level5,   20,         A_level_f

            ADD     B_sum,      A_level_f,  B_sum

            LDB     *++B_Zptr[B_run],       B_cnum
            ADD     B_Zptr,    1,           B_Zptr

[!B_eob_err]STH A_level_f, *B_outi++[B_cnum]

[!B_eob_err]B       loop
```

2.8.1 Implementation Notes and Special Requirements

The implementation notes and special requirements are the same for the C64x and the C62x, with the following exceptions:

- The zigzag matrices no longer need to be aligned at a 64-byte boundary, since illegal memory accesses to the output array are now prevented by predication of the store instruction, rather than circularly addressing the zigzag matrix (which was not possible in the C62x version because eob_err was only set *after* the store already occurred.)
- The bitstream buffer variables word1 (next_wptr-2) and word2 (next_wptr-1) are no longer reconstructed from the bitstream buffer since the larger register file of the C64x allows us to keep the variables in registers. Therefore, no special considerations have to be given to double buffering schemes.

2.9 Performance Summary

The C62x hand-optimized assembly implementations have the following cycle counts:

intra VLD routine: 10 * (S – CB) + 57 * CB + 15 * NCB + 68

inter VLD routine: 10 * S + 48 * CB + 15 * NCB + 59

By comparison, the cycle counts for the C64x implementations are:

intra VLD routine: 10 * (S – CB) + 55 * CB + 15 * NCB + 33

inter VLD routine: 10 * S + 37 * CB + 15 * NCB + 32

Where S is the number of symbols in the macroblock, CB the number of coded blocks and NCB the number of not-coded blocks (CB+NCB must be 6). For example: If one block is not coded and the coded blocks contain 75 symbols, then S = 75, CB = 5, and NCB = 1. This results in the following cycle counts:

IMPLEMENTATION	CYCLES
C62x intra	1068
C62x inter	1064
C64x intra	1023
C64x inter	982

The C64x implementations are slightly faster than those of the C62x, which is mainly due to the larger register files. The C64x versions also have the advantage that they have fewer restrictions (see section 2.8.1).

For the C62x implementation, the length tables, len and len_c, each require a 9-bit index (5 for NORM, plus 4 extra bits) with 8-bit-wide entries, which result in 512 x 8 bits = 512 bytes. The run-level table requires a 10 bit index (5 for len, plus 5 extra bits) with 16-bit wide entries. However, since the length (len) can at most be 17 only the first 576 entries are required, which results in 576 x 16 bits = 1152 bytes.

For the C64x implementation, the run-level tables are the same as for the C62x; however, both length tables have 128 more entries, thus adding 256 bytes to each set of look-up tables. The total memory requirement for one set of look-up tables, therefore, is 2432 bytes, and 2 x 2432 bytes = 4864 bytes for both tables, B–14 and B–15.

2.10 Performance Considerations for Cached Architectures

With carefully designed decoder implementations and an efficient memory system, overhead from program and cache misses for the VLD routines can be minimized. Table 7 shows estimates of stall cycles caused by L1D cold-cache misses for each data structure accessed during decode of one macroblock on C64x. As can be, the total amount of data which has to be kept in the cache is 2744 bytes, which fits well in the L1D cache of both the C6211 and C64x.

The VLD look-up tables were already designed such that codes are ordered according to decreasing probability. Thus, codes with a high probability are located next to one another in memory, reducing the likelihood of cache misses. Furthermore, the following optimizations can be performed in context of an MPEG-2 decoder:

Process multiple macroblocks at a time to reuse cached data as often as possible. The cold-cache cache misses, as shown in Table 7, will then only occur for the first macroblock. For the next macroblocks, most of cached data can be reused resulting in much fewer cache misses. This type of processing also allows the program code of the routine to be kept in the program cache (L1P), reducing the number of cache misses further.

Allocate data arrays contiguously in memory, to minimize conflict misses.

Align each data array at a L1D cache line size (64 bytes for C64x), to minimize compulsory misses.

Table 7. Estimated Number of Stall Cycles Caused by Cold-Cache Misses for Each Data Structure During Decode of One Macroblock on C64x

Data Array	Size in L1D/Bytes	Worst Case Stall Cycles	Best Case Stall Cycles
Quantization matrix	128	12	12
Inverse zigzag matrix	64	6	6
VLC length look-up table	640	60	6
VLC run-level look-up table	1152	108	6
Mpeg2v context structure	36	6	6
DC predictors (intra only)	12	6	6
Bitstream data	288	6	6
Output data	384	36	36
Stack	40	12	12
TOTAL	2744	252	96

NOTE: It is assumed here that the data arrays are allocated in L2 SRAM.

For example, a current full MPEG-2 decoder implementation on the C64x, using the VLD routines discussed in this application report obtains processing cycles for VLD with less than 13% overhead compared to the theoretical CPU formulas. Table 7 shows some details for the different frame types. The benchmark for flat memory was obtained on the simulator, while the benchmark with cache was obtained on the C6414 target.

Table 8. VLD Performance Comparison on C64x for Flat Memory and Cached-Memory Systems (test bitstream gi_9.m2v)

Frame	Flat Memory/10^6 Cycles	With Cache/ 10^6 Cycles	Degradation %
I	2.964	3.294	11.13
P	1.297	1.452	11.95
B1	0.612	0.690	12.75
B2	0.904	1.018	12.61
B3	0.941	1.018	8.18

3 H.263 Variable-Length Decoding

This section describes an implementation of variable-length decoding for video decoding systems using the ITU H.263 standard [2].

3.1 Decoding Process

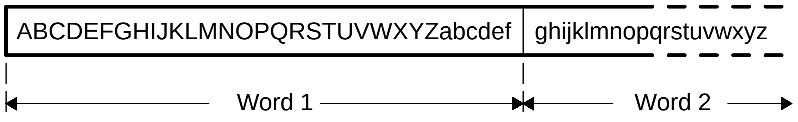
An intra DC is present for blocks of macroblocks type 3 or 4, and is coded as an 8-bit fixed-length word.

The most commonly occurring transform (AC) coefficients are coded with three dimensional VLC codes giving a LAST, RUN, and LEVEL value. If LAST = 0, then there are no more non-zero coefficients in the block. RUN denotes the number of successive zeros preceding the coded non-zero coefficient LEVEL. A sign bit, s, is appended to the VLC code, denoting a positive level if s = 0 and a negative level if s = 1.

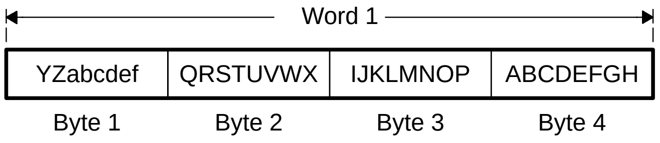
The remaining LAST, RUN, LEVEL combinations are coded using an 22-bit escape sequence consisting of 7 bits for the VLC code, denoting the start of an escape sequence, 1 bit for LAST, 6 bits for RUN, and 8 bits for LEVEL.

3.2 Bitstream Management

The bitstream is processed in word-sized chunks, using word-wide loads from the bitstream. Therefore, it is necessary to store the bitstream in ”word order”. For example, consider the bitstream shown in Figure 6 (part “a”), (left-most bits are earliest in time).



(a) Bitstream in Processing Order



(b) Bitstream as Stored in Memory

Figure 6. H.263 Bitstream Organization

The VLD routine will read the bitstream a word at a time. On a little endian CPU, this implies that the bitstream has to be stored in memory as shown in Figure 6 (part “b”).

When the VLD starts, the first two valid words are read into the variables w0 and w1. Depending on the bit position specified in the variable bit, an appropriate set of 32 bits are obtained from these two words. This is done by shifting as follows:

vld_buf = SHL(w0, (32 – bit)) | SHR(w1, bit)

Note that this relies on the C6x behavior of returning zero for shift amounts in the range of 32...63.

Advancing the bitstream forward is done by subtracting the code length from the bit-position bit, which is the number of available valid bits in w1. If an insufficient number of bits are available to form an updated vld_buf, a new word, w2, is read from the bitstream buffer.

```
w2 = buf[2];
bit –= nbit;

if (bit <= 0)
{
  w0   = w1;
  w1   = w2;
  bit += 32;
  buf++;
}

vld_buf = SHL(w0, (32 – bit)) | SHR(w1, bit);
```

The VLD routine will read up to two full words beyond the final position of the bitstream word pointer, as part of its look-ahead.

3.3 Length Computation

The VLD routine classifies VLC codes according to various ranges of bits, as shown in Figure 7. These ranges have been chosen by examining the different codes contained in the H.263 VLC table. The bit ranges indicated are extracted in to the variables f5, f7, f9, idx2, and idx1.

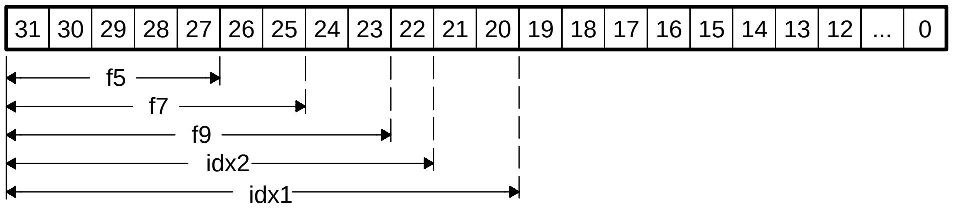


Figure 7. Classification of a H.263 VLC Code Word

The length of the VLC code is determined with a pair of look-ups. The length look-up table is a set of values constructed from code words that have been divided into two classes:

class 1 code words:	`00000xxxxx...00011xxxxx`	f5< 4
class 2 code words:	`001xxxxx...1xxxxx`	f5 > = 4

For class 1, the first 9 bits (f9), and for class 2, the first 5 bits (f5) of the code word are used as the table index. It can be seen that under class 1 there are 64 entries (0 to 63), since the first 3 bits of f9 are always zero. Under class 2, there can be a set of 32 codes, of which only 28 are accessed since f5 >= 4. The index f5 is offset by 60 so that look-up occurs over the table entries 64 to 91.

The complete length look-up table is defined as follows:

```
unsigned char len[92] =
{
     0, 12, 11, 11, 11, 11, 11, 11, 12, 12, 13, 13, 22, 22, 22, 22,
    11, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
    10, 10, 10, 10, 10, 10,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,
     9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,
     8,  8,  7,  7,  7,  7,  7,  6,  6,  6,  5,  5,  3,  3,  3,  3,
     3,  3,  3,  3,  4,  4,  4,  4,  5,  5,  5,  5
};
```

The code which performs the look-up is:

```
f5   = vld_buf >> 27;         /* First 5 bits.                    */
f9   = vld_buf >> 23;         /* First 9 bits.                    */

nbit = len[f9];               /* Codes 00000xxxxx...00011xxxxx     */
if (f5>=4) nbit = len[60+f5]; /* Codes 001xxxxx...1xxxxx           */
```

The length look-up shall be illustrated in two examples.

1. The VLC code word at index 50 of the VLC table in the H.263 standard text is 0000 1110 1s (run 19, level 1). The code falls under class 1. When the 9 left-most bits are examined, this gives us 29. The 29th entry in the length table gives us the length of the code word as 10.
2. Similarly, the code at index 40 is 0010 110s (run 10, level 1). This comes under class 2 and hence the 5 left-most bits are extracted to give an index of 5. The code lengths for these class of codes are stored with an implicit offset of 60, and hence, this code's length is actually stored at location 65. The 65th entry in the length table is 8, and corresponds to the length of the code.

3.4 Last-Run-Level Decoding

For the LAST-RUN-LEVEL look-up, the VLC codes have been grouped into three classes as follows:

1. If f5 is less than 4, then we have a code in the range 00000xxxxx to 00011xxxxx. For 00000xxxxxxx codes, idx1 is used as index. Since f5 is zero, idx1 has 7 significant bits. Therefore, the look-up occurs over the range of 0...127. For 00001xxxxx thru 00011xxxxx codes, idx2+64 is used as the look-up index. For this range of codes idx2 >= 64, and has 7 significant bits. Therefore, the look-up occurs over the range [128, 191].
2. If f5 is not less than 4, then we need to look at the first 7 bits to determine what code we have. For codes 001xxxx through 1xxxxxx codes, f7 + 176 is used as the table index. We know that f7 > = 32, therefore, the look-up will occur over the range [198,303].
3. If f7 equals 3, we have an escape code of the form, 0000011LRRRRRRMMMMMMMM, where L is the LAST flag, RRRRRR is the run, and MMMMMMMM is the signed level.

The actual code table contains 304 entries to get a multiple of 8. Each entry in the table is 16 bits wide, and has the following format:

0	0	LAST	RUN	LEVEL
16	15	14	13..8	7..0

The program code to perform the LAST-RUN-LEVEL look-up is as follows:

```
f7   = vld_buf >> 25;       /* First 7 bits.                  */
idx2 = vld_buf >> 22;       /* First 10 bits.                 */
idx1 = vld_buf >> 20;       /* First 12 bits.                 */

lrl = tab_[f7];             /* Codes: 001xxxx ... 1xxxxx      */
idx = idx1;                 /* Codes: 00000xxxxx              */
if (f5) idx = 64 + idx2;    /* Codes: 00001xxxxx .. 00011xxxxx  */
if (f5<4) lrl = tab[idx];   /* Codes: 00000xxxxx .. 00011xxxxx  */
```

```
last  = (lrl << 16) >> 30;
run   = (lrl << 19) >> 26;

l_tmp = (lrl << 25) >> 25;        /* NOTE: Unsigned Level       */
s_tmp = (vld_buf << (nbit – 1));  /* Extract sign.              */

sign  =  s_tmp < 0;               /* Set ’sign’.                */
if (sign)   l_tmp = –l_tmp;       /* Apply sign to level.       */
level =  l_tmp;                   /* Set ’level’.               */
```

For example, code 0000 0000 110s, last 0, run 0, level 11:

The first 5 bits from the left are zero, i.e., f5 = 0. Consequently, idx1 is used as table index. In this case idx1 will be either 13 or 14, depending on the sign bit. Entries 13 and 14, therefore, contain the same value which is 0x000b.

3.5 Escape Sequence

Whenever a last-run-level value that is not contained in the tables is obtained, a 22-bit escape sequence is used. The format for this escape sequence is as follows:

ESCAPE-CODE	LAST	RUN	LEVEL
7 bits	1 bit	6 bits	8 bits

The escape code is: 0000 011.

An escape sequence is present if f7, the first 7 bits, is 0x03. To shorten the recurrence path, the run and level values are decoded speculatively, assuming that this is an escape sequence. If this is not true, then the values will be conditionally replaced by the table look-up values as described above.

```
last  = ((unsigned)vld_buf <<  7) >> 31;
run   = ((unsigned)vld_buf <<  8) >> 26;
level = ((  signed)vld_buf << 14) >> 24; /* NOTE: Signed Value! */
sign  = level < 0;
```

3.6 Inverse Zigzag and Dequantization

Inverse zigzag and dequantization has been integrated into the VLD routine and is described next. De-quantization in H.263 is specified as follows:

|REC| = Quant * (2 * |Level| + 1) if Quant = ’odd’

|REC| = Quant * (2 * |Level| + 1) – 1 if Quant = ’even’

The terms are expanded out to:

|REC| = (Quant * 2) * |Level| + Quant   if Quant = “odd”

|REC| = (Quant * 2) * |Level| + Quant – 1   if Quant = “even’”

and finally,

|REC| = Q * |Level| + dq

where Q = 2 * Quant and dQ = Quant for odd Quant or dQ = Quant –1 for even Quant. Both Q and dq are passed to the routine as arguments.

After calculation of |REC| the sign is added:

REC = sign(Level) * |REC|.

Note that, in the program code, the variable level is already signed. Therefore, dq has to be made signed too. Copying of dq into dqs is done so that the negation of dq on the sign bit leaves dq untouched.

```
dq_s = dq;
if (sign) dq_s = –dq_s;
levelQ = level * Q + dq_s;
```

The run value is used to access the zigzag table to determine where to store the reconstructed level in the IDCT array. If, in case of an error, this access goes past the 63 values in the zigzag table, an error flag is set, and this term is not stored. The EOB is also set when a last value of 1 is decoded to denote end of block. In this case, the VLD ends and returns a value of 0, indicating that a valid block was decoded.

```
zz += 1 + run;;
idx = *zz;
err |= (zz > zz_end);
if (!err) idct[idx] = levelQ;
```

3.7 Error Detection and Behavior

Error detection is performed by detecting invalid Huffman codes and invalid run values which send the run-length decode outside the current 8x8 block. No test is made, however, to determine if the bitstream input underflows. The user is advised to place a sentinel bitstream at the end of the VLD buffer to prevent buffer underflow in the case of an error.

The VLD starts reading the zigzag table at the second entry (index 1) and, under normal circumstances, will not read past index 63. However, if an invalid bitstream is encountered, indices as high as 126 may be accessed.

Speculative reads are performed to each of the following arrays, as shown in Table 9.

Table 9. VLD Speculative Reads

Table/Array	Valid Range (bytes)	Accessed Range (bytes)
Zigzag table	0..63	0..127
Length table	0..91	0..511
Coef/Run table	0..607	0..607

The user should ensure that valid (e.g., CPU-readable) memory exists in the required ranges past the end of each table.

Alternately, to ensure that these reads do not reduce cache effectiveness (C6211) and do not access invalid memory, the following table layout is recommended, as it causes all speculative reads to land inside other valid table entries:

Bytes	0 .. 63	“zz”	Zig-Zag table
Bytes	64 .. 159	“len”	Huffman-code Length table
Bytes	160 .. 767	“tab”	Huffman-code Coefficient/Run Table

3.8 C Model Code

The complete C source code for the routine that performs a VLD on the AC terms of an intra or all terms of inter-coded 8x8 block for H.263, as discussed above, is presented here. Dequantization, run-length decode, and zigzag reordering are performed on each coefficient before it is written to the IDCT array. The IDCT array has to be set to zeros by the calling function. The function accepts a pointer to the bitstream buffer to be decoded, and returns the IDCT array for one block. Further, state variables are returned that allow chaining the routine, by calling it multiple times to perform VLD for a complete macroblock.

```
#ifdef _TMS320C6X
# define SHL(x,y) ( (x) << (y) )
# define SHR(x,y) ( (x) >> (y) )
#else
# define SHL(x,y) ( ((y) & 32) ? 0 : ((x) << (y)) )
# define SHR(x,y) ( ((y) & 32) ? 0 : ((x) >> (y)) )
#endif

int  h263_vld_cn
(
    unsigned int            *restrict *restrict bufPtr,
    int                     *restrict bitPtr,
    const unsigned short    *restrict tab,
    const unsigned char     *restrict len,
    short                   Q,
    int                     dq,
    const unsigned char     *restrict zz,
    short                   *restrict idct,
    int                     inter
)
{
    /* ================================================================= */
    /*  unsigned               **bufPtr,   Pointer to VLD buffer-word pointer.  */
    /*  int                    *bitPtr,    Pointer to VLD buffer-bit pointer.   */
    /*  const unsigned short   *tab,       Pointer to VLC Coeff/Run table.      */
    /*  const unsigned char    *len,       Pointer to VLC Length table.         */
    /*  short                  Q,          Quantization value.                  */
    /*  int                    dq,         Quantization/Mismatch control value. */
    /*  const unsigned char    *zz,        Pointer to zig-zag scan table.       */
    /*  short                  *idct       8x8 IDCT Coeff block. (Output)       */
    /*  int                    inter       0: intra, 1: inter-mode              */
    /* ================================================================= */

    unsigned                *restrict buf    = *bufPtr;
    int                               bit    = *bitPtr;
    const unsigned short    *restrict tab_   = &tab[176];
    const unsigned char     *restrict len_   = &len[60];
    const unsigned char     *restrict zz_end = &zz [63];
```

```
int                     EOB     = 0;
int                     err     = 0;
unsigned                w0 = buf[0], w1 = buf[1], w2;
unsigned                vld_buf;

unsigned                f5, f7, f9, idx, idx1, idx2, c4, nesc, nbit;
unsigned                run, last, lrl;
int                     level, sign, l_tmp, s_tmp, levelQ;
int                     dq_s;

/* ------------------------------------------------------------------------ */
/*  Initialize vld_buf according to the current bit position.               */
/*  Note:  This relies on >> and <<, returning 0 for shift amounts          */
/*  in the range 32..63.  Use macros to emulate this behavior on            */
/*  devices which do not behave this way.                                   */
/* ------------------------------------------------------------------------ */

vld_buf = SHL(w0, (32 – bit)) | SHR(w1, bit);

zz –= inter;

/* ------------------------------------------------------------------------ */
/*  Main VLD processing loop.                                               */
/* ------------------------------------------------------------------------ */
do
{
    /* -------------------------------------------------------------------- */
    /*  Grab portions of the bitstream and fire off table lookups.          */
    /*  See comments above for explanations of each of these variables.     */
    /* -------------------------------------------------------------------- */

    f5   = vld_buf >> 27;       /* First 5 bits.                            */
    f7   = vld_buf >> 25;       /* First 7 bits.                            */
    f9   = vld_buf >> 23;       /* First 9 bits.                            */
    idx2 = vld_buf >> 22;       /* First 10 bits.                           */
    idx1 = vld_buf >> 20;       /* First 12 bits.                           */

    c4   = f5 < 4;              /* First 5 < 4: 00000xxxxx...00011xxxxx */

    /* -------------------------------------------------------------------- */
    /*  Try to determine the length of the VLC code.  Flag an error         */
    /*  if we look up an invalid length.                                    */
    /* -------------------------------------------------------------------- */

    nbit = len[f9];             /* Codes 00000xxxxx...00011xxxxx            */
    if (!c4) nbit = len_[f5];   /* Codes 001xxxxx...1xxxxx                  */

    if (!nbit) err |= 0x2;      /* Error to 2 if length == 0.               */

    /* -------------------------------------------------------------------- */
    /*  Speculatively decode an ESCape code.  Do this whether or            */
    /*  not we actually have an ESC to reduce the recurrence.               */
    /* -------------------------------------------------------------------- */

        last  = ((unsigned)vld_buf <<  7) >> 31;
        run   = ((unsigned)vld_buf <<  8) >> 26;
        level = ((  signed)vld_buf << 14) >> 24; /* NOTE: Signed Value!     */
        sign  = level < 0;

        nesc = f7 – 3;              /* Test for ESCape:  f7 == 0000011      */
```

```
/* ------------------------------------------------------------------- */
/*  Look up the last/run/level values for the VLC, assuming this       */
/*  was _not_ an ESC.                                                   */
/* ------------------------------------------------------------------- */

if (nesc)
{
    lrl = tab_[f7];             /* Codes: 001xxxx ... 1xxxxx           */
    idx = idx1;                 /* Codes: 00000xxxxx                   */
    if (f5) idx = 64 + idx2;    /* Codes: 00001xxxxx .. 00011xxxxx     */
    if (c4) lrl = tab[idx];     /* Codes: 00000xxxxx .. 00011xxxxx     */

    last  = (lrl << 16) >> 30;
    run   = (lrl << 19) >> 26;

    l_tmp = (lrl << 25) >> 25;        /* NOTE: Unsigned Level          */
    s_tmp = (vld_buf << (nbit – 1));  /* Extract sign.                 */

    sign  =  s_tmp < 0;               /* Set ’sign’.                   */
    if (sign)   l_tmp = –l_tmp;       /* Apply sign to level.          */
    level =  l_tmp;                   /* Set ’level’.                  */
}

/* ------------------------------------------------------------------- */
/*  Perform inverse quantization, and add the appropriate              */
/*  reconstruction constant.                                            */
/* ------------------------------------------------------------------- */

dq_s = dq;
if (sign) dq_s = –dq_s;
levelQ = level * Q + dq_s;


/* ------------------------------------------------------------------- */
/*  Advance the bitstream forward by subtracting off the code’s        */
/*  length.  We speculatively load the potential next word in          */
/*  the bitstream.                                                      */
/* ------------------------------------------------------------------- */

w2 = buf[2];

bit –= nbit;

if (bit <= 0)
{
    /* --------------------------------------------------------------- */
    /*  If we have < 32 valid bits, move the bitstream by one word.    */
    /* --------------------------------------------------------------- */

    w0   = w1;
    w1   = w2;
    bit += 32;
    buf++;
}
```

```
        /* ------------------------------------------------------------------ */
        /*  Extract the top 32 bits from the bitstream for the next           */
        /*  iteration.                                                        */
        /* ------------------------------------------------------------------ */

        vld_buf = SHL(w0, (32 – bit)) | SHR(w1, bit);

        /* ------------------------------------------------------------------ */
        /*  Update the zigzag pointer according to the run–length that        */
        /*  was decoded, and read in the IDCT array index value.              */
        /* ------------------------------------------------------------------ */

        zz += 1 + run;;
        idx = *zz;

        err |= (zz > zz_end);

        if (!err) idct[idx] = levelQ;

        /* ------------------------------------------------------------------ */
        /*  We have hit End of Block if we have seen a symbol with LAST       */
        /*  set, or if we have seen at least one error.                       */
        /* ------------------------------------------------------------------ */

        EOB = last + err;

    } while (!EOB); /* Loop until end–of–block. */

    /* ---------------------------------------------------------------------- */
    /*  Store out our updated bit and word pointers.                          */
    /* ---------------------------------------------------------------------- */

    *bufPtr = buf;
    *bitPtr = bit;

    /* ---------------------------------------------------------------------- */
    /*  Return and report our error status.                                   */
    /* ---------------------------------------------------------------------- */

    return err;
}
```

3.9 Assembly Implementations

The assembly implementations follow very closely the C model presented and described previously.

The performance of the C62x hand coded assembly routine is:

cycles = 12 * N + 34

where N is the number of encoded symbols of a block. For instance, for N = 15, the routine takes 214 cycles.

The performance of the C64x hand-coded assembly routine is:

cycles = 12 * N + 24

For instance, for N = 15, the routine takes 204 cycles.

4 JPEG Variable-Length Decoding

The routine presented here performs variable-length decoding of the AC DCT coefficients. Decoding of the DC coefficient, run-length expansion, and dequantization are not included [4].

4.1 Bitstream Management

The function accepts a packed buffer of bytes containing the VLC-coded symbols. The management of the buffer is illustrated in Figure 8. The pointe,r data_ptr, points to the first byte in the data stream that is not completely contained within data_word. The variable bit_position contains the count of bits that are in the data_word, but which are before data_ptr in the bitstream. In the example shown in Figure 8, we have already consumed 11 bits of the bitstream, and are calling the routine to decode the remaining bits. The x denote bits that have already been consumed from the bitstream.

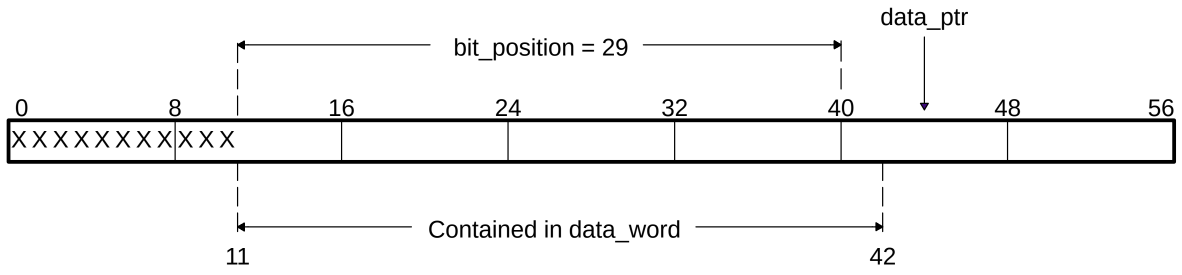


Figure 8. JPEG VLD Bitstream Management

In this case, data_word contains bits 11 through 42 of the bitstream, and data_ptr points to the byte containing bits 40 through 47 of the bitstream. In this setting, bit_position would contain 29, since three of the bits in data_word overlap with bits in the byte pointed to by data_ptr. In all circumstances, bit_position must be between 25 and 32 on entry to the routine, and all 32 bits of data_word must be valid bits from the bitstream.

On exit, bit_position, data_ptr, and data_word values are updated to reflect the current position in the bitstream, to allow chaining to other VLD routines that share this interface.

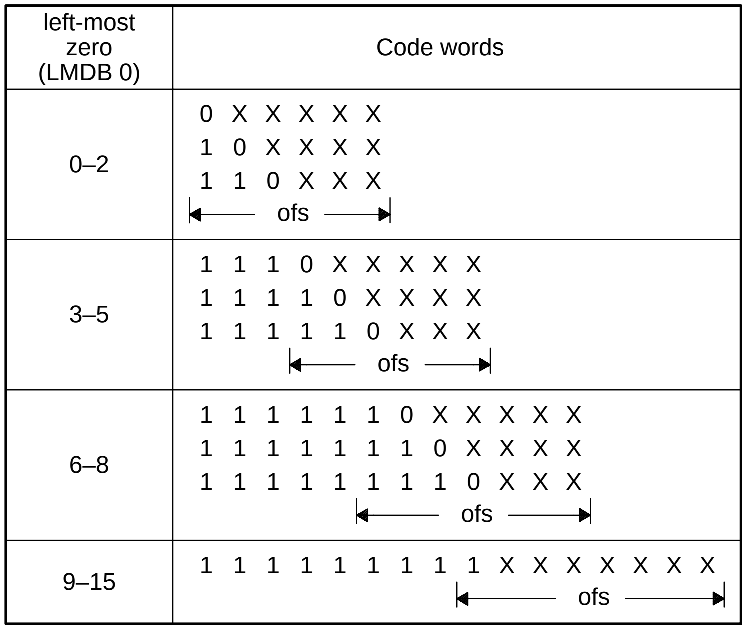
4.2 Look-up Table Layout

The variable-length code words in a JPEG stream have the following general format:

1 1 1 1 0 c c c c c l l l l l

That is, leading 1”s, a zero, some additional code bits *c* (which complete the Huffman code), followed by the level, *l*, which this symbol encodes. The JPEG VLD look-up tables are ordered according to Huffman code words so that the left-most zero value, which is obtained through the instruction LMBD 0, can be used effectively as an index into the table.The LMBD instruction returns the bit position to where a first bit reversal occurs in a register.

Commonly occurring tables (such as tables K.5 and K.6 in the JPEG standard) have groups of codes which cluster naturally into four separate left-most zero ranges, as shown in Figure 9.



left-most zero (LMDB 0)	Code words
0–2	0 X X X X X 1 0 X X X X 1 1 0 X X X |← ofs →|
3–5	1 1 1 0 X X X X X 1 1 1 1 0 X X X X 1 1 1 1 1 0 X X X |← ofs →|
6–8	1 1 1 1 1 1 0 X X X X X 1 1 1 1 1 1 1 0 X X X X 1 1 1 1 1 1 1 1 0 X X X |← ofs →|
9–15	1 1 1 1 1 1 1 1 1 X X X X X X X |← ofs →|

Figure 9. Structure of JPEG Code Words

This grouping lends itself to a look-up table which contains four sub-tables — two using a 6-bit offset, one using a 7-bit offset and a fourth using an 8-bit offset. Additionally, the 6-bit sub-tables each have 8 unused entries due to the slight overlap between left-most zero ranges. The ranges marked *ofs* in Figure 9 denote the bits that are extracted for use as a table offset into the look-up table. An extra leading 1 is extracted for two of the sub-tables, to simplify table-offset calculations.

As a result, the VLD look-up tables are laid out as shown in Figure 10 for efficient look-ups.

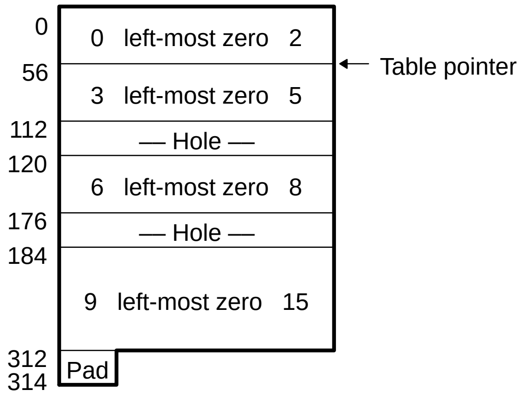


Figure 10. JPEG VLD Look-Up Table Layout

There are three such tables packed together (run, size, and shift) into one super-table. A 2-byte padding (*pad*) is placed between these three tables to allow three parallel look-ups to proceed without bank conflicts. The total super-table size is 940 bytes (312 x 3 + 4 bytes for padding).

To save time and space, the code corresponding to EOB is stored in one of the holes in the table. Specifically, offsets 112 ... 113 in the super-table hold the 16-bit EOB code, and offset 116 holds the actual length of the EOB code in bits. Although 32 bits of space are actually reserved to hold the EOB code, valid JPEG Huffman codes are no longer than 16 bits.

When decoding, the sub-table within the run/size/shift tables is selected based on the left-most zero of the register containing the top of the bitstream. This is found using the _lmbd() intrinsic on data_word. The lmbd value is calculated from the bitstream, and the table offset is computed from the significant bits to the right of the left-most zero, as shown in Figure 10.

Once calculated, the same offset is used to look up the run, size, and shift values for the Huffman code in parallel. The shift value is essentially another way of describing the code length.The size and code-length values are used for extracting the encoded level from the bitstream. Once extracted, level and run are stored in an interleaved array of runs and levels for later run-length decoding.

4.3 Implementation

4.3.1 Initialization

Three table pointers are used. They are run_tbl, sze_tbl and shf_tbl. The pointer run_tbl points to the end of the run table and uses negative offsets. The two pointers sze_tbl and shf_tbl point to the start of the size and shift tables and use positive offsets. The variables run and level are interleaved pointers to the run_level array that is used to store run and level values. They are always offset one bank from each other to avoid bank conflicts. The current bit position is contained in bit_pos_val and is between 25 and 32 at the start of the VLD routine (see also Figure 8). The variable used_bytes_val is used to keep count of the number of bytes consumed from the buffer. At the start of the program the initial value from the pointer used_up_bytes is read and passed to it. It accumulates to this value the number of bytes consumed during this run of the VLD and writes back the result in used_up_bytes before exit. This is used to chain several stages of the VLD routine. The variable data_word_val contains the first 32 bits from which the VLD parsing will begin (see also Figure 8).

The pointer, data_ptr, points to a buffer that contains the next set of bytes to be fetched from the bitstream buffer. The variable number_val is used to calculate the number of run-level pairs found for this run of VLD. This is used to convey state information for chaining VLD's.

```
tbl             =  ac_table_luma–>run_ptr;
run_tbl         =  ac_table_luma–>run_ptr + 56;
sze_tbl         =  ac_table_luma–>sze_ptr + 314 + 56;
shf_tbl         =  ac_table_luma–>shf_ptr + 628 + 56;

run             =  run_level;
level           =  run_level +1;

EOB_flag        =  ac_table_luma–>eob_flag;
bit_pos_val     =  *bit_pos_vld ;
used_bytes_val  =  *used_up_bytes;
data_word_val   =  *data_word;

data_pointer    =  *data_ptr;
numberval       =   0;
num_coeffs      =   0;

eob_byte0       =  tbl[eob_ofs]     << 24;
eob_byte1       =  tbl[eob_ofs + 1] << 16;

EOB_CODE        =  ( eob_byte0 + eob_byte1);
EOB_LEN         =  tbl[eob_ofs + 4];
EOB_MASK        =  ~(~0U >> EOB_LEN);
```

The variable, num_coeffs, maintains a running sum of the run + 1 values decoded. If this reaches 63, then the VLD decode process is exited without an EOB code actually occurring in the bit-stream. In this case, the EOB flag is forcibly set to 1 so that the VLD terminates. If this sum exceeds 63, then this is a case of an incorrect JPEG bitstream. In this case, the standard does not recommend any particular course of action. The approach taken by this routine is to continue treating this as a normal JPEG bitstream. The VLD routine forces the EOB flag to 1 and treats the case of num_coeffs > 63 identially to the case where num_coeffs = 63, except that it returns an error code.

The variables eob_byte0 and eob_byte1 are used to read the EOB code stored in offsets 112 and 113. This gives a 16-bit EOB code. These values are left justified and added together to form EOB_CODE. EOB_LEN is the length of EOB_CODE and is stored in offset 116. EOB_MASK is used to prepare a mask which has 1's in the EOB_LEN most-significant bits. For instance, for the K-5 table with EOB of length 4, EOB_ MASK is 0xF000 0000.

4.3.2 Decoding Loop

First, the EOB_flag is detected by applying EOB_MASK to data_word_val. If the result is the same as EOB_CODE, the EOB_flag is set. Then, the next 5 bytes starting at data_pointer are speculatively fetched in temporary registers, temp0 to temp4, and shifted such they can be later combined into one 40-bit value. The variable, numberval, is incremented to indicate that one run-level pair has been detected.

```
while (!EOB_flag)
{
    EOB_flag = ((data_word_val & EOB_MASK) == EOB_CODE );

    temp0       = data_pointer[0];
    temp1       = data_pointer[1];
    temp2       = data_pointer[2];
    temp3       = data_pointer[3];
    temp4       = data_pointer[4];

    temp0l      = ((uLong) temp0 << 32);
    temp1l      = ((uLong) temp1 << 24);
    temp2l      = ((uLong) temp2 << 16);
    temp3l      = ((uLong) temp3 << 8);
    temp4l      = ((uLong) temp4 << 0);

    numberval++;
```

Next, offset calculation is performed speculatively for the 4 sub-tables in the super table, according to the scheme in Figure 9. This is done using _extu and shifts, and then choosing amongst these offsets by the lmbd value. Once the offset has been figured run, size and length of the code word can be looked up.

```
    t02_ofs = data_word_val >> 26;           // lmbd value l: 0 <= l <= 2
    t35_ofs = _extu(data_word_val, 3, 26);   // lmbd value l: 3 <= l <= 5
    t68_ofs = _extu(data_word_val, 5, 25);   // lmbd value l: 6 <= l <= 8
    t9x_ofs = _extu(data_word_val, 8, 24);   // lmbd value l: >= 9

    ofs = t35_ofs;

    lmbd_val = _lmbd (0, data_word_val);
```

```
        if (lmbd_val < 3) ofs = t02_ofs – 56;
        if (lmbd_val > 5) ofs = t68_ofs;
        if (lmbd_val > 8) ofs = t9x_ofs;

        run_val   = run_tbl[ofs];
        size_val  = sze_tbl[ofs];
        len_cw    = shf_tbl[ofs];
```

Note that all tables are organized such that the same offset can be used. If the lmbd value is between 0 and 2, a negative offset is prepared by subtracting from 56, since the table pointer is pre-offset to 56 (see Figure 10). Table offset t02_ofs lies between 0 and 55 and will be folded to the range –56 to –1. Offset t35_ofs lies between 0 and 56 to cover the table entries from 56 to 112. Similar properties hold for t68 and t9x offsets.

Now the level is being calculated. First, the number of bits of the Huffman portion of the code word, len_val, is determined by subtracting the total length of the code word, len_cw, from the number of extra bits, size_val. After left-shifting the data word by len_val, temp_ac contains the extra bits left-justified (and other bits from the next code word). Neg_AC is the sign of the level and is 0 for positive, and 1 for negative numbers. Note that the MSB of the extra bits is 0 for negative levels and 1 for positive levels. By performing a signed shift by 31 of the extra bits, the result is sign extended and is –1 for MSB = 1 and 0 for MSB = 0. If then 1 is added, Neg_AC contains the correct 2's complement sign bit.

To get the level, temp_ac is shifted right to have the extra bits right-justified, and converted to 2s complement representation if NegAC is 1.

```
        len_val   = len_cw – size_val;
        temp_ac   = data_word_val << len_val;
        Neg_AC    = 1 + ((signed)temp_ac >> 31);
        level_val = _shr(temp_ac,(32-size_val)) – (Neg_AC << size_val ) +
Neg_AC;
```

For example, level_val of -3 is coded as 00b with size_bits = 2. This would be decoded as 00b –100b + 1b = 11b, i.e., 0 – 4 + 1 = –3. A level_val of 3 is code as 11b with size_bits = 2. This would be decoded as 11b – 0b + 0b = 11b, i.e., 3 – 0 – 0 = 3.

The run and level values are now stored into memory, and the corresponding pointers incremented appropriately. The variable num_coeffs keeps track of the total run. The current bit position is updated by subtracting the length of the codeword bit_pos_val, and the data word is advanced to the next code word.

Next, the number of complete bytes are determined so they can be inserted to update data_word_val with new bitstream data. For instance, if the present bit_pos_val is 9, then 2 complete bytes, and 7 bits of the 3rd byte, can be inserted. These 7 bits constitute the overlap between data_word_val and dat_ptr.

```
        *run      = run_val;
        *level    = level_val;
        run      += 2;
        level    += 2;

        num_coeffs         += (run_val+1);
        bit_pos_val        –= len_cw ;
        data_word_val     <<= len_cw ;
        num_bytes_insert_val = (32-bit_pos_val)>>3;
        data_pointer        += num_bytes_insert_val;
        used_bytes_val      += num_bytes_insert_val;
```

The bytes temp0l...temp4l that were previously fetched are used to form a 40-bit word, tempword. A right-shift brings this word into the position where it can be pasted into data_word_val. Since a 40-bit shift is being used, 8 is added to bit_pos_val to compute the shift amount. The loop iterates until an EOB is detected or the total run becomes >= 63.

```
        tempword       = temp0l+temp1l+temp2l+temp3l+temp4l;
        data_word_temp = (unsigned int) _shrl(tempword, 8+bit_pos_val);
        data_word_val |= data_word_temp;
        bit_pos_val   += (num_bytes_insert_val*8);

        if (num_coeffs >= 63)    EOB_flag = 1;
```

At the end of the loop, the number of run-level pairs (num_of_rles), the current bit position within the data word (bit_pos_vld), and the number of bytes used so far in the buffer (used_up_bytes) are stored out. In addition, store the existing data_pointer and the state word of data_word_val. All these variables are used as state variables to perform chaining of VLD stages from one invocation of the routine to the other. As part of error checking, the VLD returns 1 if the number of coefficients is greater than 63, indicating an error, and 0 if the decoded block is a valid.

```
    *num_of_rles = numberval – 1;
    *bit_pos_vld = bit_pos_val;
    *used_up_bytes= used_bytes_val;
    *data_word    = data_word_val;
    *data_ptr     = (unsigned char *)data_pointer;

    return (num_coeffs > 63);
```

4.4 C Model Code

```
#ifdef _TMS320C6X
# define _shr(a,b)  ((a) >> (b))
# define _shrl(a,b) ((a) >> (b))
# ifndef _INLINE
#  define _INLINE inline
# endif
typedef unsigned long uLong;
#else
# define _shr(a,b)  (((b) & 32) ? 0 : (a) >> (b))
# define _shrl(a,b) (((b) > 40) ? 0 : (a) >> (b))
# ifndef _INLINE
#  define _INLINE
# endif
typedef unsigned long long uLong;   /* must be >= 40 bits in length */
#endif

int  ac_vld_decode_cn
(
    unsigned char       *restrict *restrict data_ptr,
    unsigned int        *restrict data_word,
    short               *restrict run_level,
    unsigned char       *restrict num_of_rles,
        unsigned char       *restrict bit_pos_vld,
        unsigned int        *restrict used_up_bytes,
        const AC_TBL_str    *restrict ac_table_luma
    )
    {

        unsigned int    Neg_AC, run_val, size_val, len_val;
        unsigned int    temp_ac, num_coeffs;
        unsigned int    numberval;
        unsigned int    temp0,temp1, temp2,temp3,temp4;
        uLong           temp0l,temp1l,temp2l,temp3l,temp4l,tempword;

        unsigned int    used_bytes_val,data_word_val;
        unsigned int    len_cw,num_bytes_insert_val;
        unsigned int    EOB_MASK,EOB_LEN,EOB_CODE;
        unsigned int    EOB_flag,data_word_temp;

        const unsigned char *restrict sze_tbl;
        const unsigned char *restrict data_pointer;
        const unsigned char *restrict shf_tbl;
        const unsigned char *restrict run_tbl;
        const unsigned char *restrict tbl;

        short *restrict run, *restrict level;

        int bit_pos_val;
        int level_val;
        int eob_ofs = 112;
        int eob_byte0,eob_byte1;
        int t02_ofs, t35_ofs, t68_ofs;
        int t9x_ofs, ofs;

        /* ------------------------------------------------------------------ */
        /*  Set the pointers to tbl, run, sze, and shf arrays                 */
        /* ------------------------------------------------------------------ */

        tbl             =  ac_table_luma->run_ptr;
        run_tbl         =  ac_table_luma->run_ptr + 56;
        sze_tbl         =  ac_table_luma->sze_ptr + 314 + 56;
        shf_tbl         =  ac_table_luma->shf_ptr + 628 + 56;

        run             =  run_level;
        level           =  run_level +1;

        EOB_flag        =  ac_table_luma->eob_flag;
        bit_pos_val     =  *bit_pos_vld ;
        used_bytes_val  =  *used_up_bytes;
        data_word_val   =  *data_word;


        data_pointer    =  *data_ptr;
        numberval       =   0;
        num_coeffs      =   0;

        eob_byte0       =  tbl[eob_ofs]     << 24;
        eob_byte1       =  tbl[eob_ofs + 1] << 16;
```

```
/* ------------------------------------------------------------------- */
/*  Prepare EOB_CODE, EOB_LEN and EOB_MASK, based on user.             */
/* ------------------------------------------------------------------- */

EOB_CODE        =  ( eob_byte0 + eob_byte1);
EOB_LEN         =  tbl[eob_ofs + 4];
EOB_MASK        =  ~(~0U >> EOB_LEN);


while (!EOB_flag)
{
    EOB_flag = ((data_word_val & EOB_MASK) == EOB_CODE );

    temp0       = data_pointer[0];
    temp1       = data_pointer[1];
    temp2       = data_pointer[2];
    temp3       = data_pointer[3];
    temp4       = data_pointer[4];

    temp0l      = ((uLong) temp0 << 32);
    temp1l      = ((uLong) temp1 << 24);
    temp2l      = ((uLong) temp2 << 16);
    temp3l      = ((uLong) temp3 << 8);
    temp4l      = ((uLong) temp4 << 0);

    numberval++;

    t02_ofs = data_word_val      >> 26;
    t35_ofs = data_word_val << 3 >> 26;
    t68_ofs = data_word_val << 5 >> 25;
    t9x_ofs = data_word_val << 8 >> 24;

    ofs = t35_ofs;

    if (data_word_val <  0xE0000000UL) ofs = t02_ofs – 56;
    if (data_word_val >= 0xFC000000UL) ofs = t68_ofs;
    if (data_word_val >= 0xFF800000UL) ofs = t9x_ofs;

    run_val   = run_tbl[ofs];
    size_val  = sze_tbl[ofs];
    len_cw    = shf_tbl[ofs];

    len_val   = len_cw – size_val;
    temp_ac   = data_word_val << len_val;
    Neg_AC    = 1 + ((signed)temp_ac >> 31);
    level_val = _shr(temp_ac,(32–size_val)) –
                (Neg_AC  <<     size_val ) + Neg_AC;

    *run      = run_val;
    *level    = level_val;
    run      += 2;
    level    += 2;

    num_coeffs         += (run_val+1);
    bit_pos_val        –= len_cw ;
    data_word_val     <<= len_cw ;
```

```
            num_bytes_insert_val = (32–bit_pos_val)>>3;
            data_pointer        += num_bytes_insert_val;
            used_bytes_val      += num_bytes_insert_val;

            tempword       = temp0l+temp1l+temp2l+temp3l+temp4l;
            data_word_temp = (unsigned int) _shrl(tempword, 8+bit_pos_val);
            data_word_val |= data_word_temp;
            bit_pos_val   += (num_bytes_insert_val*8);

            if (num_coeffs >= 63)    EOB_flag = 1;
        }

        *num_of_rles  = numberval – 1;
        *bit_pos_vld  = bit_pos_val;
        *used_up_bytes= used_bytes_val;
        *data_word    = data_word_val;
        *data_ptr     = (unsigned char *)data_pointer;

        return (num_coeffs > 63);

    }
```

4.5 Assembly Implementations

The assembly implementations follow very closely, the C model presented and described previously. To further improve performance, the recurrence path has been shortened by maintaining two copies of the bitstream buffer register, data_word_val, and pre-incrementing bit_pos by 8. Having two copies of data_word_val permits working with one instance while the other one is being already updated simultaneously for the next code word. Further, a copy of data_pointer is made on the opposite data path, to allow loads from bitstream buffer to parallelize.

```
MV              A_data_word_val,     B_data_word_cpy
ADD             A_bit_pos_val,       8,                A_shift_new
MV              A_data_pointer,      B_data_pnt_cpy
```

For the C64x implementation, a non-aligned word load, LDNW, is used to read four bytes at once. The fifth byte is loaded using LDBU. Since the bitstream is left-justified, the loaded non-aligned word has to be endian-swapped. This is done by the use of the SWAP4 and ROTL instructions.

The performance of the hand-optimized version on C62x is:

cycles = 17 * N + 45

where N is the number of run-level pairs decoded. For example, to decode a block with N = 15 run-level pairs, 300 cycles are required.

On C64x the routine is somewhat faster because of the use of LDNW:

cycles = 14 * N + 27

For a typical JPEG image with 15 symbols/block, 237 cycles/block are required.

5 Review of Implementation Techniques

This section reviews the implementation techniques used for the three implementations discussed in the previous sections. The problem areas may be grouped into inherent problems, data structures, and processor resources. Table 10 shows a summary of all the problem areas and roughly rates their impact on speed, look-up table memory size, and cache performance (if applicable).

Table 10. Overview of Problem Areas for VLD Implementations and Their Impact (o = no impact, + = low impact, ++ = high impact)

		Speed	LUT Memory Size	LUT Cache Performance	Comments
Inherent problems	Recurrence in bitstream advance	++	o	o	Restricts initiation interval
	While-loop	++	o	o	Increased initiation interval due to possibly additional instructions/registers required to preserve live-outs
Data structures	Bitstream management	++	o	o	Affects recurrence path
	Length LUT layout and indexing	++	++	++	How fast the length can be looked up affects recurrence path
	Event LUT layout and indexing	+	++	++	Memory-speed trade-off. Grouping together more probable symbols increases cache performance.
Processor resources	Functional unit bottlenecks	++	o	o	.S unit is likely to be bottlenecked. Use alternative instructions.
	Bank conflicts	+	o	o	LUT access is spatially irregular. Do not schedule memory accesses in parallel.
	Cross Path stalls (C64x)	++	o	o	Avoid. Increases recurrence path length
	Register pressure	+	o	o	High, due to large number of live-outs

5.1 Inherent Problems

These type of problems arise through the very nature of the variable-length decoding process.

5.1.1 Recurrence Path

Since the start of the next VLC code will be only known after the length of the current one has been determined and the bitstream advanced accordingly, the start of the next iteration has to be delayed by at least the number of cycles it takes to perform these two steps. In other words, length computation and bitstream advance constitute a recurrence path which limits the minimum initiation interval (II) that can be achieved. Typically, the resource bound does not become an issue unless dequantization is integrated into the decoding loop. Therefore, the shorter the recurrence path, the better the initiation interval that can be achieved.

5.1.1.1 Length Computation

The length of the code can either be determined through table look-up or calculation. In both cases, a unique characteristic is extracted from the code (e.g., leading number of zeros plus an additional number of bits) which can be either used as an index into the table, or for computing the length. Which method requires fewer cycles depends on the nature of the look-up and the underlying architecture. Each case has to be analyzed individually, but generally one can say that if the look-up table is relatively large, and the characteristics that can be extracted from the bitstream do not correlate well with the run-level values, calculation may be more expensive than look-up.

5.1.1.2 Bitstream Advance

The way the bitstream is buffered and advanced directly impacts on the length of the recurrence path. The following schemes have been discussed in the earlier sections:

- Byte-wise update of 40-bit register buffer (for JPEG)
- Word-wise update of 32-bit register buffer (H.263)
- Word-wise update of 32-bit register double buffer (MPEG-2 C62x)
- Word-wise update of 40-bit/32-bit register double buffer (MPEG-2 C64x): In order to gain from extended buffer, it must be able to hold at least 2 look-up VLC codes or 1 look-up VLC code plus the VLC part of an escape sequence.

Figure 11 illustrates the different bitstream buffering schemes. The arrows indicate with which data a register is updated. The schemes are ordered in increasing efficiency. The advantage of double-buffering schemes over single-buffering schemes is that the update of the word-aligned buffers from memory is removed from the recurrence path. For instance, the scheme used for H.263 does not ensure that the bit-aligned buffer can be fully updated from the word-aligned buffer without first updating the word-aligned buffer itself from memory. For example, assuming that 28 bits of the word-aligned buffer have already been consumed, then 32 bits consisting of bits 3 to 0 from the left word, and bits 31 to 4 from the right word are copied into the bit aligned buffer. At this point the word aligned buffers cannot be updated since the bits in the left word buffer may still be required, in case, say, the length of the VLC code was only 2 bits (then bits 1 to 0 and 31 to 3 need to be copied into the bit aligned buffer). If the length of the code was, say, 22 bits, then we have now consumed 28+22 = 50 bits, leaving only 64 – 50 = 12 fresh bits in the word aligned buffer which is not enough to update the bit aligned buffer. Therefore, this scheme requires to update the word aligned buffer before the bit aligned buffer can be updated, adding to the length of the recurrence path. The scheme used for JPEG suffers from a similar drawback. Additionally, the bitstream buffer in memory is accessed in byte-size elements which are then used to construct a 40-bit word aligned buffer and so adding considerably to the cycle count of the recurrence path. The advantage of this scheme however is that the bitstream is not required to be stored in little endian 32-bit elements.

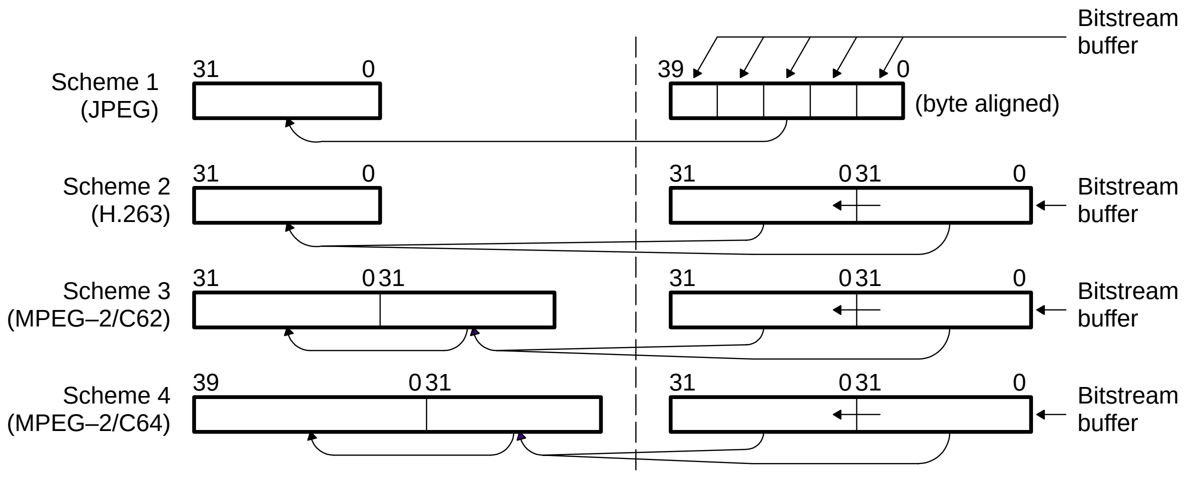


Figure 11. Different Bitstream Buffering Schemes

5.1.1.3 Cross Path Stalls (C64x)

On the C64x, cross path stalls may add to the length of the recurrence path. It should be checked if scheduling on one side only, and thus completely avoiding cross paths provides a better solution. Further, it may be beneficial to consider alternative bitstream management schemes as described above.

5.1.2 While Loop

The core of any VLC decoding routine is a loop with unknown trip count. Since multiple iterations are overlapped, typically, some instructions have to be predicated once an EOB is detected. This is required to preserve variables which would otherwise be modified by the following iteration, which should not have been executed, and would not reflect the correct state. The later the EOB is detected the more operations of the following iteration(s) that will have already been executed. Those operations which change the state have to be predicated and prevented from execution if EOB was detected. Other methods to preserve the correct state include moving relevant registers into save registers, saving them on the stack (beware bank conflicts), and reconstructing them after loop exit.

5.2 Look-Up Table Layout and Indexing

Another issue is the size of the look-up tables. If the VLC code word were directly used as an index into a table, the table would consume too much memory for most applications. For instance, in MPEG-2, the length of a code can be up to 16 bits. This would require a table with 65,536 entries, whereas the actual number of codes used would be only 113. Using such a table, one could always extract the first 16 bit from the start of the current code word, regardless the actual length of the code word and use these as an index into the table to determine length and run/level. Since we may also be using bits from the next code word, a requirement is that regardless of these bits, the table must always return the same value corresponding to the current code word. Example: If the maximum code length is 4 bits and the code 01 is to be decoded as 7, then the entries 0100, 0101, 0110 and 0111 all have to contain 7, since we always pick 4 bits. In other words, the two last bits of the index must be “don’t cares” (01XX). This is the reason for the many redundant table entries. The codes are constructed such that they can be decoded uniquely. In the case of the example, no other code would start with 01. Usually, a table can be split up into a number of smaller tables by identifying common characteristics of code words (such as number of leading 0's or 1's), or, in the extreme case, the value may even be calculated in which case no table is required at all.

There is a trade-off between the size of the table and how fast we can compute the index (or value). The number of cycles it takes to determine the code length affects the overall speed of the algorithm. If a full size table is used, decoding can be done very quickly, since all we have to do is to use a fixed number of bits as index into the table, read the length and advance the bitstream. If the table is split up, then there are some extra calculations necessary to extract characteristics of the code word. The more sub-tables we use, the smaller the memory requirements, but the more complicated it may get to derive the index from the code word. Completely calculating the value may be most computationally intensive. However, the effect on the performance is not clear cut. On the C6000, a memory access executes over more pipeline stages than other operations. For calculating the value, more operations may be required, but due to parallel functional units, it may require less cycles than a memory access combined with a few extra calculations.

The above description is a general strategy of how to trade off speed and memory size. However, the actual implementation has to be done according to the specifics of a VLC table.

5.3 Processor Resources

5.3.1 Funtional Unit Bottlenecks

Since many operations for variable-length decoding are performed on bit level, the usage of shift instructions is typically quite high, which causes a bottleneck on the .S unit. To achieve a better balance of functional unit usage, shift instructions may be replaced with alternative operations, such as MPY x, 8, y instead of SHL x, 3, y, assuming that x is a 16-bit number.

5.3.2 Bank Conflicts

Accesses to the quantization matrix, inverse zigzag matrix, and the look-up tables do not follow a regular pattern, making it difficult to predict possible bank conflicts. Therefore, scheduling memory accesses in parallel should generally be avoided. An exception is the length/32-length look-up, where the offset between those memory accesses is known, and the two tables can be positioned in memory such that no bank conflicts occur. Also, note that in the case of the C6211, bank conflicts are not an issue, since memory is dual-port.

5.3.3 Register Allocation

Typically, the usage of conditional registers is quite high and can prevent a successful register allocation. Therefore, the aim should be to have a similar usage-availability ratio for both partitions. Overall register pressure can be very high on the C62x, especially due to the number of variables describing the current state of the bitstream buffer that have to be preserved. Instead of moving these state variables into save registers, some may have to be saved on the stack or reconstructed after loop exit from the variables that could be saved. Register pressure can also be reduced by trying to minimize the number of constants and address pointers kept in registers. For instance, some tables may be combined in an interleaved fashion.

5.4 Integrating Run-Length Expansion, Inverse Zigzag and Dequantization

Inverse zigzag and dequantization can be integrated into the decoding loop to make it more efficient. In the case of the MPEG-2 VLD presented in this application report, the additional operations fit in the already available instruction slots, providing inverse zigzag and dequantization virtually for free.

5.5 Error Behavior

It is important to prevent stores to unpredictable locations in memory due to errors in the bitstream. Store instructions should either be predicated or indexing made circular. For instance, the inverse zigzag array can be made circular, which prevents both illegal accesses to outside the inverse zigzag array due to invalid run values, and also to the output array, since it uses the content of inverse zigzag array as an index.

6 Conclusion

This application report discussed efficient VLD implementations for JPEG, H.263, and MPEG-2. Further, optimization strategies and techniques were explained in great depth, enabling the reader to use these techniques on other, possibly proprietary, VLD algorithms. The design of VLD look-up tables and code optimization of the algorithm can be very challenging. However, with the methods and techniques presented in this application report, very good performance can be achieved for software-based implementations on the TMS320C6000. An advantage of a software implementation is that all existing video and image coding standards as well as proprietary algorithms can be supported, and that end-user equipment can be upgraded easily.

7 References

1. MPEG-2 Video Standard (ISO/IEC 11172 Part 2: Video)
2. ITU-T H.263, Video Coding for Low Bit Rate Communication
3. Joan L. Mitchel and Willliam B. Pennebaker, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1992.
4. *TMS320C6000 JPEG Implementation* (SPRA704).
5. *TMS320C6000 CPU and Instruction Set Reference Guide* (SPRU189).
6. *TMS320C62x Image/Video Processing Library Programmer’s Reference* (SPRU400).
7. *TMS320C64x Image/Video Processing Library Programmer’s Reference* (SPRU023).