Entropy Decoding on TriMedia/CPU64

Mihai Sima[1,2], Evert-Jan Pol[2], Jos T.J. van Eijndhoven[2],
Sorin Cotofana[1], and Stamatis Vassiliadis[1]

[1] Delft University of Technology, Department of Electrical Engineering,
Mekelweg 4, 2628 CD Delft, The Netherlands,
{M.Sima,S.D.Cotofana,S.Vassiliadis}@et.tudelft.nl
[2] Philips Research Laboratories, Department of Information and Software Technology,
Professor Holstlaan 4, 5656 AA Eindhoven, The Netherlands,
{evert-jan.pol,jos.van.eijndhoven}@philips.com

Abstract. The paper describes a software implementation of an MPEG–compliant Entropy Decoder on a TriMedia/CPU64 processor. We first outline entropy decoding basics and TriMedia/CPU64 architecture. Then, we describe the reference implementation of the entropy decoder, which consists mainly of a software pipelined loop. On each iteration, a set of look-up tables partitioning the Variable-Length Codes (VLC) table defined by the MPEG standard are accessed in order to retrieve the *run-level* pair, or detect an *end-of-block* or *error* condition. An average of 21.0 cycles are needed to decode a DCT coefficient according to this reference implementation. Then, we focus on software techniques to optimize the entropy decoding software pipelined loop. In particular, we propose a new way to partition the VLC table such that by exposing the loop prologue to the compiler, testing each of the *end-of-block* and *error* conditions within the prologue becomes superfluous. This is based on the observation that either an *end-of-block* or *error* condition will never occur within the first table look-up. For the proposed implementation, the simulation results indicate that an average of 16.9 cycles are needed to decode a DCT coefficient. That is, our entropy decoder is more than 20% faster than its reference counterpart.

1 Introduction

The introduction of digital audio and video was the starting point of multimedia because it enabled audio and video, as well as text, figures, and tables, to be used in a digital form in a computer and be held in the same manner. However, digital audio and video require a tremendous amount of information bandwidth unless compression technology is used, which in turn calls for a large amount of processing. For example, National Television Systems Committee (NTSC) resolution MPEG-2 [1] decoding requires more than 400 MOPS, and 30 GOPS are required for encoding.

TriMedia/CPU64 is a VLIW core targeted for real-time processing of multimedia streams [2]. Although its processing power allows significant processing of video data, the VLIW core itself was intended to be integrated on-chip with a set of hardwired co-processors which can perform other tasks with stringent real-time requirements in parallel. An example of such co-processor is the Variable-Length Decoder (VLD) [3].

One of the drawbacks of the hardwired solution is the lack of flexibility, since a different full-custom circuit is needed for each particular task. Software programmability ensures that a single device can be applied in a range of different products and can adapt to quickly evolving standards in the media domain. Therefore, a software solution which can provide the needed performance is always preferred to the hardware solution.

When the application exhibits data and instruction-level parallelisms, TriMedia/CPU64 has proved significant speed-up over previous TriMedia families [4]. However, the speed-up is not so high when parallelism is not available. Entropy decoding [5, 6] consists of Variable-Length Decoding (VLD) followed by a Run-Length Decoding (RLD), both VLD and RLD being sequential tasks. Due to data dependency, entropy decoding is an intricate function on TriMedia, since a VLIW architecture must benefit from instruction level parallelism in order to be efficient.

An entropy decoder implementation on TriMedia/CPU64 which can decode a Discrete Cosine Transform (DCT) coefficient in 21 cycles has been proposed by Pol [7]. The VLD is implemented as a repetitive look-up into the Variable-Length Codes (VLC) table defined by MPEG standard, where each iteration analyzes a fixed-size chunck of bits. When a coefficient is completely decoded, a *run-level* pair is generated, otherwise an offset into the VLC table is generated. By employing software pipeline optimization techniques, run-length decoding for the previous decoded symbol is carried out simultaneously with the variable-length decoding of the current symbol.

In this paper we demonstrate that significant improvement over the reference solution is possible if four optimizations are used:

1. partitioning the VLC table in such a way that by exposing the prologue of the software pipeline loop to the compiler, an *end-of-block* symbol or *error* will never be encountered within the prologue;
2. using an extended *barrel-shift* TriMedia-specific operation;
3. storing the lookup tables in such way that all the fields (*run*, *level*, *table_offset*, etc) are each located within the boundaries of a byte. This way, the extraction of each and every such field can be done in a single cycle by TriMedia–specific operations;
4. using variable chunck size, in order to reduce the total size of the tables.

The testing database for our entropy decoder consists of a number of pre-processed MPEG conformance strings from which all the data not representing DCT coefficients have been removed. Therefore, such strings include only *run-level* and *end-of-block* symbols. The simulations carried out on a TriMedia/CPU64 cycle accurate simulator indicate that 16.9 cycles are needed to decode a DCT coefficient with the proposed implementation. That is, our entropy decoder is 20% faster than its reference counterpart.

As an evaluation of the absolute performance of the entropy decoder we propose, we would like to mention some figures claimed by our competitors: 33 cycles per coefficient which exploits SIMD–type operations of a Pentium processor with MultiMedia eXtension (MMX) are claimed by Ishii *et al.* [8], and 26 cycles per coefficient on an TMS320C80 media video processor are claimed by Bonomini *et al.* [9].

The paper is organized as follows. Section 2 gives some background information concerning MPEG compression standard and TriMedia/CPU64 architecture. Entropy decoder implementation issues are presented in Section 3. The experimental framework and results are presented in Section 4. The final section concludes the paper.

2 Background

The MPEG standard [6, 10] uses a large number of compression techniques to decrease the amount of data. Data compression is the reduction of redundancy in data representation, carried out to decrease data storage requirements and data communication costs.

A typical video codec system is presented in Figure 1 [5, 6]. The lossy source coder performs filtering, transformation (such as Discrete Cosine Transform (DCT), subband decomposition, or differential pulse-code modulation), quantization, etc. The output of the source coder still exhibits various kinds of statistical dependencies. The (loseless) entropy coder exploits the statistical properties of data and removes the remaining redundancy after the lossy coding.

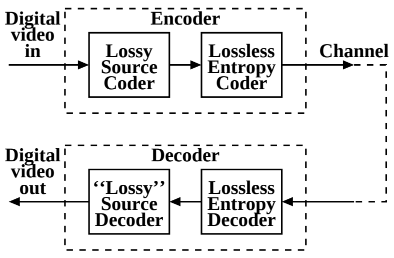

Fig. 1. A generic video codec.

In MPEG, the couple DCT + Quantization is used as a lossy coding technique. The DCT algorithm processes the video data in blocks of 8×8 pixels, decomposing each block into a weighted sum of amplitudes of 64 spatial frequencies. At the output of DCT, the data is also organized as 8×8 blocks of coefficients, each coefficient representing the contribution of a spatial frequency for the video block being analyzed. Since the human eye cannot readily perceive high frequency activity, a quantization step is then carried out. The goal is to force as many DCT coefficients as possible to zero within the boundaries of the prescribed video quality. Then, a zig-zag operation transforms the matrix into a vector of coefficients which contains large series of zeros. This vector is further compressed by an Entropy Coder which consists of a Run-Length Coder (RLC) and a Variable-Length Coder (VLC). The RLC represents consecutive zeros by their run lengths; thus the number of samples is reduced. The RLC output data are composite words, referred to as *symbols*, which describe a *run-level* pair. The *run* value indicates the number of zeros by which a (non-zero) DCT coefficient is preceeded. The *level* value represents the value of the DCT coefficient. When all the remaining coefficients in a vector are zero, they are all coded by the special symbol *end-of-block*. Variable length coding is a mapping process between *run-level*/*end-of-block* symbols and *variable length codewords*, which is carried out according to a set of tables defined by the standard. Not every run-level pair has a variable length codeword to represent it, only the frequent used ones do. For those rare combinations, an *escape* code is given. After an *escape* code, the run- and level-value are coded using fixed length codes.

In order to achieve maximum compression, the coded data does not contain specific guard bits separating consecutive codewords. As a result, the decoding procedure must recognize the *code-length* as well as the symbol itself. Before decoding the next symbol, the input data string has to be shifted by a number of bits equal to the decoded code length. These are recursive operations that generate true-dependencies.

Subsequently, we will focus on the entropy decoding, i.e., on the operation inverse to entropy coding. We will briefly present some theoretical issues connected to variable-length decoding and run-length decoding.

2.1 Entropy Decoder

In MPEG, the entropy decoder consists of a **Variable-Length Decoder** (VLD) followed by a **Run-Length Decoder** (RLD). The input to the VLD is the incoming bit stream, and the output is the decoded symbols. As depicted in Figure 2, a VLD is a system with feedback, whose loop contains a *Look-Up Table* (LUT) on the feed-forward path and a *bit parser* on the feedback path. The LUT receives the variable-length code itself as the address [11] and outputs the decoded symbol (*run-level* pair or *end_of_block*) as well as the codeword length, *code_length*. In order to determine the starting position of the next codeword, the *code_length* is fed back to an accumulator and added to the previous sum of codeword lengths, ***accumulated code_length***. The bit parsing operation is completed by the *barrel-shifter* (or *funnel-shifter*) which shifts out the decoded bits.

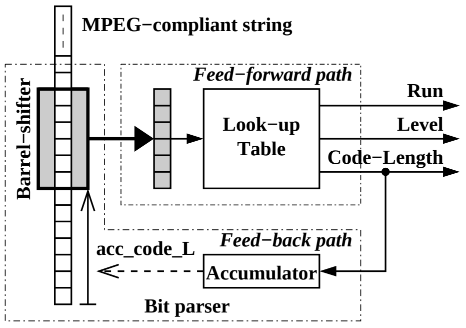

Fig. 2. Variable-length decoding principle.

In connection with the hardware complexity, we would like to note that the longest codeword excluding Escape has 17 bits. Therefore, the LUT size reaches $2^{17} = = 128$ K words for a direct mapping of all possible codewords. Regarding the performance of a variable-length decoder, it is worth mentioning that the throughput of a VLD is bounded by a value inverse to the latency of the loop [12].

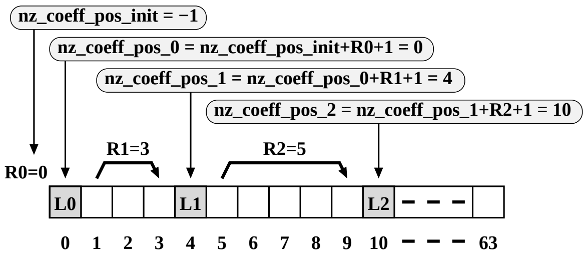
nz_coeff_pos_init = −1
nz_coeff_pos_0 = nz_coeff_pos_init+R0+1 = 0
nz_coeff_pos_1 = nz_coeff_pos_0+R1+1 = 4
nz_coeff_pos_2 = nz_coeff_pos_1+R2+1 = 10
R0=0
R1=3
R2=5
L0
L1
L2
0 1 2 3 4 5 6 7 8 9 10 63

Fig. 3. Run-length decoding principle.

Conceptually, for each *run-level* pair returned by the VLD, the run-length decoder outputs the number of zeros specified by the *run* value and then pass the *level* through. In a programmable processor–based platform, a way to optimize this process is to fill in an empty vector with *level* values, L, at positions defined by *run* values, as depicted in Figure 3: the position of a non-zero coefficient, `nz_coeff_pos`, is computed by adding the *run* value, `R`, and an '`1`' to the position of the previous non-zero coefficient. This common strategy has been widely used in previous work [1, 7, 13] and will be used subsequently, too.

In connection with the software implementation of the entropy decoder we propose, we would like to mention that both VLD and RLD are sequential tasks. Consequently, entropy decoding is an intricate function on TriMedia, since a VLIW processor must benefit from instruction-level parallelism in order to be efficient.

The next subsection will outline some elements of the MPEG-2 standard related to variable-length decoding.

2.2 MPEG-2–compliant Variable-Length Decoding

MPEG-2 defines four tables for encoding the DCT coefficients: B12, B13, B14, and B15 [1]. Which table is used depends on the type of image – intra (I) or non-intra (NI), luminance (Y) or chrominance (C) – and a bit-field, `intra_vlc_format`, in the macroblock header, as shown in Table 1. In general, this means that a single stream uses all tables, and the tables can be switched per macroblock and/or block.

`intra_vlc_format`			0	1
I	DC coefficient	Y	B12	B12
		C	B13	B13
	AC coefficient		B14	B15
NI	1st & subsequent coefficient		B14	B14

Table 1. Selection of VLC tables

In the decoding process of DCT coefficients, there are a few exceptional cases to be dealt with:

1. **The DC coefficient** for intra macroblocks: this coefficient is encoded through the B12/B13 tables, depending on the block type: luminance or chrominance.
2. **Escape**: escape code is 6 bits long, followed by 6 bits run and 12 bits signed level.
3. **end-of-block**: this is a 2 or 4 bit code, depending on the `intra_vlc_format` bit.

Apart from these cases, the decoding follows "normal" coding rules. The maximum code-length is 16 bits plus a sign bit. A code determines a *run* and a *level* value. A variable-length code is followed by a sign bit that indicates the sign of the *level* value.

We conclude this section with a review of the TriMedia/CPU64 VLIW core.

2.3 TriMedia/CPU64 architecture

TriMedia/CPU64 is a simulated processor designed to be used in the development process of future 64-bit VLIW cores. Its architecture features a very rich instruction set optimized for media processing. Specifically, TriMedia/CPU64 is a 64-bit 5 issue-slot VLIW core, launching a long instruction every clock cycle [2]. It has a uniform 64-bit wordsize through all functional units, register file, load/store units, on-chip highway and external memory. Each of the five operations in a single VLIW instruction can in principle read two register arguments and write one register result every clock cycle. In addition, each operation can be optionally guarded with the least-significant bit of a fourth register, in order to allow for conditional execution without branch penalty. The architecture supports subword parallelism; for example, operations such as *additions/subtractions*, *shuffle*, *elementwise multiplexing*, on eight 8-bit unsigned integers (`vec64ub`), or on four 16-bit signed integers (`vec64sh`) are possible. Super-operations, which occupy two adjacent slots in the VLIW instruction, and map to a double-width functional unit are also supported. The current organization of the TriMedia/CPU64 core is presented in Figure 4.

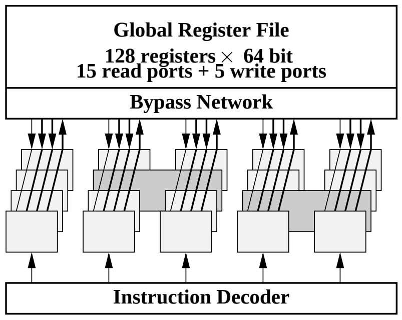

Fig. 4. TriMedia/CPU64 organization.

3 Entropy decoder implementation

According to the reference implementation [7], the VLD is implemented as a repeated table-lookup. Each lookup analyzes a fixed size chunk of bits (for example, `LOOKUP_ADDRESS_WIDTH` = 6 or 8) and determines if a valid code was encountered or some more bits need to be decoded. In any case, the number of consumed bits ranging from the smallest variable-length code to the chunk size is generated. In case of a valid decode, i.e., *hit*, a *run-level* pair is generated, or an *escape* or *end_of_block* flag is set. If a *miss* is detected, i.e., more bits are needed for a valid decode, an offset into the VLC table for a second- or third-level lookup, *table_offset*, is generated. This process of signaling an incomplete decode and generating a new offset may be repeated a number of times, depending on the largest variable-length code and chunk size.

The following basic stages can be discerned in the reference implementation of the entropy decoder on TriMedia/CPU64:

1. **Initializations**.
2. **Barrel-shift the VLC string** according to the *accumulated code-length* value.
3. **Table look-up** (look-up address computation, table look-up proper). The table look-up returns a 32-bit word containing all the fields mentioned at Stage 4.
4. **Field extraction**: *run*, *level*, *code_length*, *valid_decode*, *end_of_block*, *escape*, *table_offset*.
5. **Update (modulo-64) the accumulated code-length**:
   acc_code_length = *acc_code_length* + *code_length*
   If an overflow has been encountered, **advance the VLC string** by 64 bits.
6. **Exit** the loop if *end_of_block* has been encountered.
7. **Handle escape** if *escape* has been encountered.
8. **Run-length decoding**: de-zig-zag, followed by filling-in an empty 8×8 matrix.
9. **Go to** Stage 2.

The Stage 8 – **run-length decoding** – is folded into the loop, such that loop pipelining is employed [7]. That is, the run-length decoding for the previous decoded symbol is carried out simultaneously with the variable-length decoding of the current symbol.

Updating the *acc_code_length* value is carried out modulo-64. The main idea is to match this process with the transfer capabilities of the 64-bit version of TriMedia. That is, a new chunk of 64 bits of information to be decoded is read on overflow. Also, we would like to emphasize that the VLC-related information is stored into the lookup table in a packed format, as 32-bit unsigned integers, as depicted in Table 2. Therefore, a sequence of masking and shifting operations are needed to extract these fields.

Table 2. The original VLC table format.

	end-of-block (stop)	escape	valid	run	level	table offset	code-length
No. of bits	1	1	1	5	8	12	4
Position	31	30	29	28-24	23-16	15-4	3-0

To make the presentation self consistent, the reference implementation of the entropy decoding routine is presented in Algorithm 1. All identifiers written with capital letters are regarded as constants. In the sequel, we will provide some additional information regarding this algorithm, highlighting efficiency-related issues.

Algorithm 1 Entropy decoder routine – reference implementation

```
 1: set-up the test-bench (store the VLC lookup table, read the VLC_string into memory, etc.)
 2:
 3: for $i = 1$ to NO_OF_MACROBLOCKS do
 4:   for $j = 1$ to NO_OF_BLOCKS_IN_MACROBLOCK do
 5:     table_offset ← FIRST_TABLE_OFFSET
 6:     nz_coeff_pos_ZZ ← 0
 7:     run ← 0
 8:     valid_decode ← 0
 9:
10:     loop
11:       barrel-shift the VLC_string with acc_code_length positions
12:       lookup_address ← the leading LOOKUP_ADDRESS_WIDTH bits from VLC_string
13:       lookup_address ← lookup_address + table_offset
14:       retrieved_32_bit_word ← VLC_table[lookup_address]
15:
16:       nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + run
17:       nz_coeff_pos ← invZZ_table[nz_coeff_pos_ZZ]
18:       $8 \times 8$_matrix[nz_coeff_pos] ← level
19:       nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + valid_decode
20:
21:       extract code_length, run, level, table_offset, escape, valid_decode, end_of_block from
          retrieved_32_bit_word
22:
23:       acc_code_length ← acc_code_length + code_length
24:       if acc_code_length $\leq 64$ and not(escape) then
25:         continue { ——————————–> go to loop}
26:       end if
27:       if end_of_block flag is raised then
28:         break { ———————————–> initiate the next for iteration (block-level)}
29:       end if
30:       if acc_code_length $\geq 64$ then
31:         advance the VLC_string by 64 bits
32:         acc_code_length ← acc_code_length - 64
33:       end if
34:       if escape flag is raised then
35:         run ← next 6 bits from VLC_string
36:         level ← next 12 bits from VLC_string
37:         acc_code_length ← acc_code_length + 6 + 12
38:       end if
39:     end loop
40:   end for
41: end for
```

The entropy decoder routine consists of a first **for** loop (lines 3–41) cycling over all macroblocks in the MPEG conformance string, a second **for** loop (lines 4–40) cycling over all blocks in a macroblock, and an inner (infinite) loop labeled **loop** (lines 10–39), cycling over all DCT coefficients in a block. The inner loop is left only when an *end_of_block* is encountered (lines 27–29).

The initializations for block-level decoding are performed at lines 5–8. Table look-up, i.e., variable-length decoding, is carried out at lines 11–13. Lines 15–18 implement run-length decoding, which, as we already mentioned, is folded into the loop in order to employ loop pipelining. Field extraction is performed at line 20. The barrel-shifting (line 11) is done on an 128-bit field, by means of a TriMedia–specific operation:

bitfunshift Rsrc_1 Rsrc_2 Rsrc_3 $\rightarrow$ Rdest_1 Rdest_2

where `Rsrc_1` and `Rsrc_2` are the two 64-bit registers storing the leading 128 bits of the VLC_string to be shifted, the `Rsrc_3` defines the shifting value, and `Rdest_1` and `Rdest_2` are the two 64-bit registers storing the 128-bit shifted field. Obviously, only the value stored into `Rdest_1` register will be used for the look-up procedure. It should be mentioned that since *acc_code_length* is updated modulo-64 (lines 30–33), at least 47 bits are available in `Rdest_1` for the next decoding iteration in the most defavorable case (this can be easily verified by assuming that *acc_code_length* = 63 at line 34).

A particular optimization technique has been used in order to keep the most likely iteration (that is when no more incoming bits from the MPEG string are needed, and none of the *escape*, *end_of_block*, and *error* conditions is raised), as short as possible. According to this technique, the *escape* flag is also set to '1' when any of the *escape*, *end_of_block*, or *error* conditions occurs. In this way, a jump to the beginning of the inner loop is taken when none of the above mentioned conditions is raised (lines 24–26). All the exceptional cases are managed after this jump: *end_of_block* at lines 27–29, modulo-64 updating and advancing the VLC string at lines 30–33, and *escape* at lines 34–38. It should be mentioned that there is no flag to indicate an *error* condition. When an *error* is encountered, *end_of_block* = 1 and *valid_decode* = 0 simultaneously. Therefore, the loop will be left because the *end_of_block* flag is set. However, it is the responsibility of the entropy decoder calling routine to detect if a valid *end_of_block* has been detected or an *error* has occured. Since this subject is beyond the goal of the paper, it will not be analyzed in the sequel.

In connection to the efficiency of the reference implementation, we would like to specify that the major drawback of the software pipeline is that only variable-length decoding for the first DCT coefficient will be performed during the first iteration, the code associated with run-length decoding being dummy. That is, the method penalty is the overhead needed to fire-up the software pipeline. Since the number of non-zero DCT coefficients in a block is rather small, ranging, for example, between 3.3 and 5.8 for non-intra macroblocks [7], the number of iterations per block is also small. Consequently, this overhead can be significantly large.

In the sequel, we will discuss the improvements that we propose with respect to decoding of non-intra macroblocks. That is, the VLC table will be the B14 table defined by the MPEG standard if we will not state otherwise.

To improve the performance of the entropy decoder, we propose the following changes in respect with the reference implementation:

- **The prologue of the pipelined loop** [14] **is exposed to the compiler**. Since the VLC table does not have "holes" in the region of short code-length coefficients (i.e., each and every entry in the VLC table in that region corresponds either to a short codeword which can be decoded in a single iteration, or to a long codeword which will be decoded in two or more iterations), there are no incoming bit combinations which do not have a meaning within the prologue. Therefore, an *error* condition will never be raised. Moreover, since an *end_of_block* symbol is not allowed for the first coefficient in a block, an *end_of_block* condition will never be encountered, too. Consequently, testing the *end_of_block* flag (lines 27–29 in Algorithm 1) within the prologue becomes superfluous and can be eliminated. For this reason, a very simple code consisting of a first-level look-up, folowed by an extraction of the *code_length, run, level, lookup_address_width, table_offset, escape, valid_decode* (and, notable, no extraction of the *end_of_block* flag) can efficiently fire-up the software pipeline.
- **Barrel-shifting is carried out by means of an extended `bitfunshift` TriMedia specific operations**.

  ```
  bitfunshift_3 Rsrc_1 Rsrc_2 Rsrc_3 Rsrc_4 → Rdest_1 Rdest_2
  ```

  The main idea is to gain flexibility over the modulo-64 operation by performing the barrel-shift operation on $3 \times 64 = 192$ bits instead of $2 \times 64 = 128$ bits. In this way, the modulo-64 operation can be postponed, since additional 64 bits are available for decoding over the standard implementation.
- **The lookup returns a 64-bit value instead of a 32-bit value**. The main idea is to store each of the *code_length, run, level, lookup_address_width* (which defines the chunk size of the next look-up), *table_offset, escape, valid_decode* (signals a hit), and *end_of_block* fields within the boundaries of a byte (that is, in an unpacked way instead of a packed one). Since extracting a byte from a 64-bit value takes only 1 cycle on TriMedia, our solution is two times faster than using a pair of masking and shifting operations required by the 32-bit approach. The cost of such approach is a double-size look-up table. It is still an open question which approach is better with respect to a particular TriMedia cache size, as the cache misses may become a bottleneck when the performance evaluation is made for a complete MPEG decoder. The new format of the VLC table format is presented in Table 3.
- **The chunk size is variable**, which leads to a more compact look-up table. According to our experiments, there are enough empty slots in the TriMedia instruction format for an entropy decoding task. Consequently, a variable chunk size does not introduce real dependencies.

In connection with the Table 3, several comments should be provided. The VLC table is a one-dimensional array of vectors, where each vector contains eight unsigned bytes. In order to keep the number of instructions as low as possible, we propose to store the sign bit of each and every codeword into the lookup table.

Table 3. The proposed VLC table format.

	code-length	run	level	table offset	lookup address width	escape	valid decode	EOB
No. of bits	8	8	8	8	8	8	8	8
Position	63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0

According to Table B14, the *level* value ranges between $-40 \cdots + 40$. Thus, 7 bits (less than 1 byte) are sufficient to represent all the values. However, precautions have to be taken to convert *level* to a signed integer after extraction (Algorithm 2).

Algorithm 2 Converting the *level* from 8-bit unsigned integer to a 16-bit signed integer

```
#define LEVEL_FIELD 5

int16 level;

retrieved_vec64ub = VLC_table[ lookup_address];
level = (int16) ub_get( retrieved_vec64ub, LEVEL_FIELD);
level = (int16)((level ≪ 24) ≫ 24); /* 32-bit processing */
```

The least significant byte has been allocated for *end_of_block* (EOB) flag. Since the TriMedia C compiler recognizes expressions of the form $(E_1\&1)$, the least significant bit of this byte is set to '1' when an *end_of_block* condition is raised. This way, the condition for leaving the loop can be written as follows:

Algorithm 3 TriMedia-specific code for testing the *end-of-block* condition

```
#define END_OF_BLOCK_FIELD 0

uint8 end_of_block;

for (;;) {
  retrieved_vec64ub = VLC_table[ lookup_address];
  end_of_block = ub_get( retrieved_vec64ub, END_OF_BLOCK_FIELD);
  if ( end_of_block & 1)
    break;
}
```

The *table_offset* field defines the partitioning of the B14 into smaller lookup tables. The B14 table has been splitted in eight tables (*first*, *second*, *third*, *forth*, *fifth*, *sixth*, *seventh*, *eighth*) which are presented subsequently. We mention that, in order to improve the readness, we preserved the order of the rows as in the MPEG standard.

VL code	Run	Level
1s	0	1
011s	1	1
0100 s	0	2
0101 s	2	1
0010 1s	0	3
0011 1s	3	1
0011 0s	4	1
0001 10s	1	2
0001 11s	5	1
0001 01s	6	1
0001 00s	7	1
0000 110s	0	4
0000 100s	2	2
0000 111s	8	1
0000 101s	9	1
0000 01	Escape	

Table 4. First table

VL code	Run	Level
10	End of Block	
11s	0	1
011s	1	1
0100 s	0	2
0101 s	2	1
0010 1s	0	3
0011 1s	3	1
0011 0s	4	1
0001 10s	1	2
0001 11s	5	1
0001 01s	6	1
0001 00s	7	1
0000 110s	0	4
0000 100s	2	2
0000 111s	8	1
0000 101s	9	1
0000 01	Escape	

Table 5. Second table

1st prefix	VL code	Run	Level
0010 0	110 s	0	5
	001 s	0	6
	101 s	1	3
	100 s	3	2
	111 s	10	1
	011 s	11	1
	010 s	12	1
	000 s	13	1

Table 6. Third table

1st prefix	VL code	Run	Level
0000 001	0 10s	0	7
	1 00s	1	4
	0 11s	2	3
	1 11s	4	2
	0 01s	5	2
	1 10s	14	1
	1 01s	15	1
	0 00s	16	1

Table 7. Forth table

1st prefix	VL code	Run	Level
0000 0001	1101 s	0	8
	1000 s	0	9
	0011 s	0	10
	0000 s	0	11
	1011 s	1	5
	0100 s	2	4
	1100 s	3	3
	0010 s	4	3
	1110 s	6	2
	0101 s	7	2
	0001 s	8	2
	1111 s	17	1
	1010 s	18	1
	1001 s	19	1
	0111 s	20	1
	0110 s	21	1

Table 8. Fifth table

1st prefix	VL code	Run	Level
0000 0000	1101 0s	0	12
	1100 1s	0	13
	1100 0s	0	14
	1011 1s	0	15
	1011 0s	1	6
	1010 1s	1	7
	1010 0s	2	5
	1001 1s	3	4
	1001 0s	5	3
	1000 1s	9	2
	1000 0s	10	2
	1111 1s	22	1
	1111 0s	23	1
	1110 1s	24	1
	1110 0s	25	1
	1101 1s	26	1
	0111 11s	0	16
	0111 10s	0	17
	0111 01s	0	18
	0111 00s	0	19
	0110 11s	0	20
	0110 10s	0	21
	0110 01s	0	22
	0110 00s	0	23
	0101 11s	0	24
	0101 10s	0	25
	0101 01s	0	26
	0101 00s	0	27
	0100 11s	0	28
	0100 10s	0	29
	0100 01s	0	30
	0100 00s	0	31

Table 9. Sixth table

1st prefix	2nd prefix	VL code	Run	Level
0000 0000	001	1 000s	0	32
		0 111s	0	33
		0 110s	0	34
		0 101s	0	35
		0 100s	0	36
		0 011s	0	37
		0 010s	0	38
		0 001s	0	39
		0 000s	0	40
		1 111s	1	8
		1 110s	1	9
		1 101s	1	10
		1 100s	1	11
		1 011s	1	12
		1 010s	1	13
		1 001s	1	14

Table 10. Seventh table

1st prefix	2nd prefix	VL code	Run	Level
0000 0000	0001	0011 s	1	15
		0010 s	1	16
		0001 s	1	17
		0000 s	1	18
		0100 s	6	3
		1010 s	11	2
		1001 s	12	2
		1000 s	13	2
		0111 s	14	2
		0110 s	15	2
		0101 s	16	2
		1111 s	27	1
		1110 s	28	1
		1101 s	29	1
		1100 s	30	1
		1011 s	31	1

Table 11. Eighth table

All eight tables are stored into memory one after another, i.e., in a concatenated way. The number of address bits for each table is related to the maximum length of the variable-length codes. That is, Tables *first* and *second* have each 8 address bits, Table *sixth* has 7 address bits, Tables *third* and *forth* have each 4 address bits, and Tables *fifth*, *seventh*, and *eighth* have each 5 address bits. Thus, the sizes of the tables are as follows:

Table	No. of address lines (*lookup_address_width*)	Size (64-bit words)	*table_offset*
first	8	$2^8 =$ 256	`0`
second	8	$2^8 =$ 256	`0x100`
third	4	$2^4 =$ 16	`0x200`
forth	4	$2^4 =$ 16	`0x210`
fifth	5	$2^5 =$ 32	`0x220`
sixth	7	$2^7 =$ 128	`0x240`
seventh	5	$2^5 =$ 32	`0x2c0`
eighth	5	$2^5 =$ 32	`0x2e0`

Table 12. Number of address lines, size, and offset for each VLC table

with a total of 768 64-bit words, which means 6 KB.

The decoding procedure can be exemplified on Figure 5. Let us suppose that the following string is to be decoded: `1000000000011000110....` The *table_offset* is initialized to `0`, that is the *first* table is being pointed to. Also, *lookup_address_width* is initialized to `8`, which means that the first 8 bits of the string, i.e., `10000000`, will be regarded as address into the *first* table. The following values are retrieved: *code_length* = 2, *run* = 0, *level* = 1, *table_offset* = `0x100`, and *lookup_address_width* = `8`. Which means that the *second* table will be accessed during the second iteration.

After shifting out the two bits decoded at the previous iteration, the leading eight bits, i.e., *00000000*, will be regarded as address, this time into the *second* table. By looking-up, *code_length* = `8`, *table_offset* = `0x240`, and *lookup_address_width* = `7`. That is, the *sixth* table will be accessed. No valid *run-level* pair has been detected.

At this moment the *accumulated_code_length* is 10. Therefore, the leading 10 bits have to be shifted out from the input string. Then, the next seven bits, i.e., *0011000*, are regarded as address into the *sixth* table. Again, no valid *run-level* pair is detected. The *code_length* = `3`, *table_offset* = `0x2c0`, *lookup_address_width* = `5`. That is, the *seventh* table will be accessed.

After incrementation, the *accumulated-code-length* = 13. After shifting out the leading 13 bits, the next five bits, i.e., *10001* are the address into the *seventh* table. The look-up procedure retrieves the following values: *code_length* = 5, *run* = `0`, *level* = `-32`, *lookup_address_width* = `8`, *table_offset* = `0x100` bypassing the *first* table. That is, all subsequent coefficients of the 8×8 block will use only the Tables *second - eighth.*

Finally, the accumulated-code-length is 18. The next eight bits to be sent as address to the *second* table are: *10xxxxxx.* An *end_of_block* symbol is detected, and the *table-offset = 0*; that is, the *first* table is to be accessed for decoding of a new block.

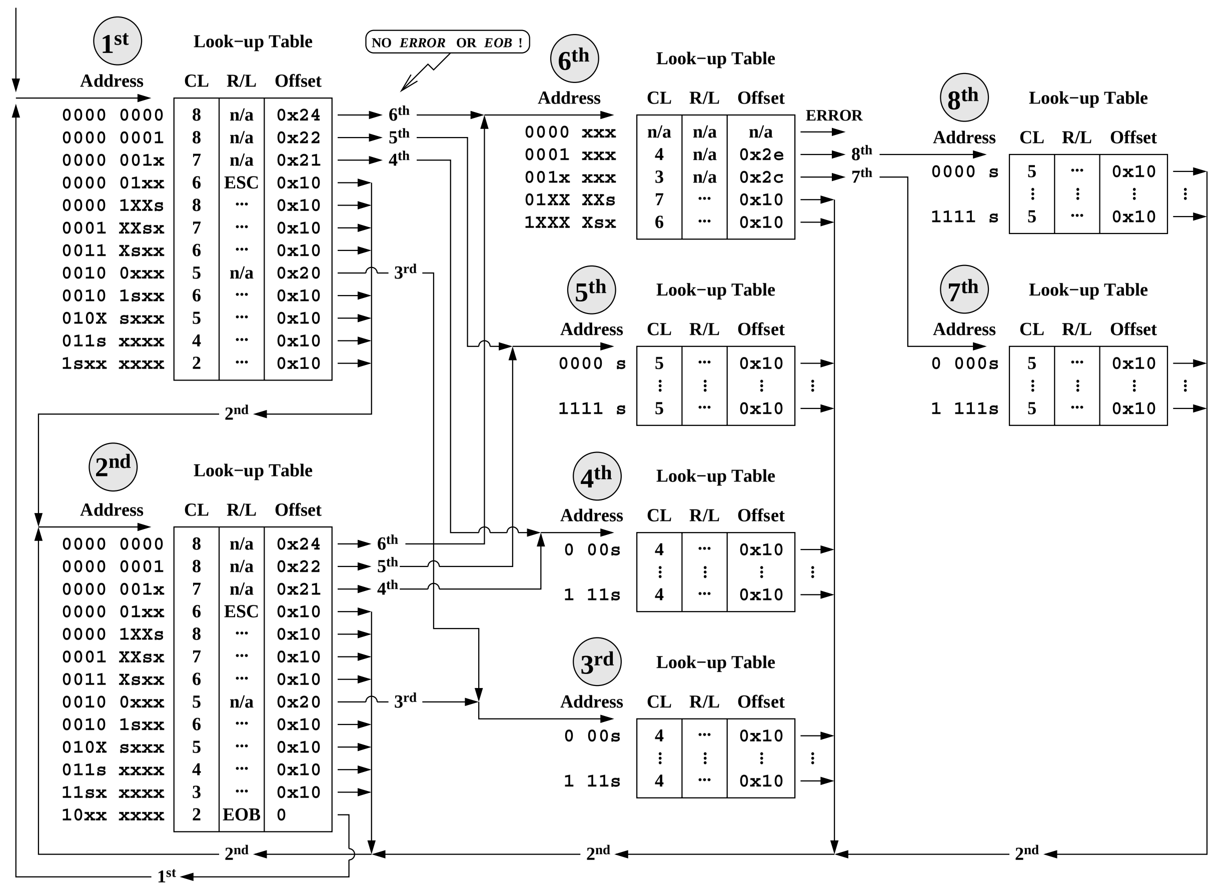

Fig. 5. The flowchart of the variable-length decoding procedure.

The entropy decoder implementation we propose is presented in Algorithm 4. As it can be observed, the prologue of the inner (infinite) loop (lines 17–45) has been exposed to the compiler (lines 4–15). Since an *end_of_block* or *error* condition will never occur on the first table lookup (line 7), testing the *end_of_block* condition during the prologue becomes superfluous and, therefore, has been eliminated.

Special considerations have to be provided with respect to modulo-64 operation. As me already mentioned, since the extended `bitfunshift` TriMedia-specific operation is used, more flexibility in postponing the modulo-64 operation is gained. Indeed, there is no such operation within the prologue. However, from the MPEG syntax point of view this is not entirely correct. Assuming that *acc_code_length* is 63 at line 36, it will become 81 at line 45. Considering that an *end_of_block* is encountered, then *acc_code_length* = 83. If this situation occurs during the decoding of the first block in a macroblock, and if the subsequent five coded blocks in the same macroblock include each an *escape* sequence followed by an *end_of_block*, then *acc_code_length* $= 83 + 5 \times 24 + 5 \times 2 = 213$, that is more than the limit of 192 bits. Fortunately, this case is not statistically relevant (we did verify it on all MPEG conformance strings mentioned in the subsequent section). Fortunately, this exceptional situation can be overcomed without much penalty by augmenting the *escape* handling code within the prologue (lines 11–15) with a modulo-64 operation.

The same strategy of exposing the prologue of the loop to the compiler can be applied for decoding of intra blocks, since an *end_of_block* can never occur during the decoding of an DC coefficient. However, special precautions have to be taken in order to deal with errors.

Finally, it should be mentioned that standard optimization techniques such as *loop unrolling* or *grafting* [15] cannot be applied, because that would introduce awkward *escape* code and/or barrel-shifting processing.

4 Experimental results

The testing database for our entropy decoder consists of a number of pre-processed MPEG conformance strings, or *scenes*, from which all the data not representing DCT coefficients have been removed. Therefore, such strings include only *run-level* and *end-of-block* symbols.

For all experiments described subsequently, the MPEG-compliant bit string is assumed to be entirely resident into the main memory. In this way, side effects associated with bit string acquisition such as asynchronous interrupts, trashing routines, or other operating system related tasks, do not have to be counted. Moreover, saving the reconstructed 8×8 matrices into memory, as well as zeroing these matrices in order to initialize a new entropy decoding task are equally not considered. Since both procedures can be considered parts of adjacent tasks, such as IDCT or motion compensation, they are subject to further optimizations at the complete MPEG decoder level. Thus, in our experiments, the run-length decoder will overwrite the same 8×8 matrices again and again. With these assumptions, the only relevant metric is the number of instruction cycles required to perform strictly entropy decoding. Therefore, the main goal was to minimize this number.

Algorithm 4 Entropy decoder routine with the prologue exposed to the compiler

```
1: for $i = 1$ to NO_OF_MACROBLOCKS do
2:   for $j = 1$ to NO_OF_BLOCKS_IN_MACROBLOCK do
3:     nz_coeff_pos_ZZ ← 0
4:     barrel-shifting the VLC_string
5:     lookup_address ← the leading FIRST_LOOKUP_ADDRESS_WIDTH bits from VLC_string
6:     lookup_address ← lookup_address + (FIRST_TABLE_OFFSET $\ll$ 4)
7:     retrieved_vec64ub ← VLC_table[lookup_address]
8:
9:     extract code_length, run, level, table_offset, lookup_address_width, escape,
       valid_decode from retrieved_vec64ub {end_of_block field is not extracted!}
10:    acc_code_length ← acc_code_length + code_length
11:    if escape flag is raised then
12:      run ← next 6 bits from VLC_string
13:      level ← next 12 bits from VLC_string
14:      acc_code_length ← acc_code_length + 6 + 12
15:    end if
16:
17:    loop
18:      barrel-shift the VLC_string
19:      lookup_address ← the leading lookup_address_width bits from VLC_string
20:      lookup_address ← lookup_address + table_offset
21:      retrieved_vec64ub ← VLC_table[lookup_address]
22:
23:      nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + Run
24:      nz_coeff_pos ← invZZ_table[nz_coeff_pos_ZZ]
25:      8 × 8_matrix[nz_coeff_pos] ← Level
26:      nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + valid_decode
27:
28:      extract code_length, run, level, table_offset, lookup_address_width, escape,
         valid_decode, end_of_block from retrieved_vec64ub
29:      acc_code_length ← acc_code_length + code_length
30:      if acc_code_length $\leq 64$ and not(escape) then
31:        continue { ——————————-> go to loop}
32:      end if
33:      if end_of_block flag is raised then
34:        break { ————————————-> initiate the next for iteration (block-level)}
35:      end if
36:      if acc_code_length $\geq 64$ then
37:        advance the VLC_string by 64 bits
38:        acc_code_length ← acc_code_length - 64
39:      end if
40:      if escape flag is raised then
41:        run ← next 6 bits from VLC_string
42:        level ← next 12 bits from VLC_string
43:        acc_code_length ← acc_code_length + 6 + 12
44:      end if
45:    end loop
46:  end for
47: end for
```

Table 13. Entropy decoding experimental results.

Scene (*.m2v)	Block type	Workload (coeff.)	Scene profile (bit/coeff.)	Reference implementation (cycle/coeff.)	Proposed implementation (cycles)	Proposed implementation (cycle/coeff.)	**Improvement**
batman	I (B15)	172,745	5.5	21.85	2,843,376	16.5	22.5 %
	NI	266,485			4,592,358	17.2	
popplen	I (B15)	47,003	5.3	20.19	777,553	16.5	17.3 %
	NI	28,069			475,326	16.9	
sarnoff2	I (B14)	80,563	5.1	21.9	1,387,489	17.2	23.3 %
	NI	36,408			577,388	15.9	
tennis	I (B14)	12,345	6.1	21.77	210,011	17.0	20.7 %
	I (B15)	120,754			1,937,808	16.0	
	NI	137,756			2,527,395	18.3	
ti1cheer	I (B15)	80,818	5.1	20.75	1,311,687	16.2	21.9 %
	NI	51,680			836,082	16.2	

The results for entropy decoder are presented in Table 13. The figures indicate the number of instruction cycles needed to decode the pre-processed MPEG string. The last column of the table specifies the relative improvement of the proposed entropy decoder versus its reference counterpart. Unfortunately, only the average number of cycles per coefficient has been disclosed for the reference implementation [7].

It is also worth mentioning that the absolute performance of the proposed entropy decoder ranges between $15.9 \div 18.3$ cycles/coeff., with the mean 16.9 cycles/coeff. This is a very good result with respect to both 33.0 cycles/coeff. needed for variable-length decoding and Inverse Quantization (IQ) on a Pentium processor with MultiMedia eXtension (MMX) claimed by Ishii et. al [8], and 26.0 cycles/coeff. achieved on an TMS320C80 media video processor by Bonomini *et al.* [9]. The additional IQ functionality considered by the referred papers is not a real concern for us, since our preliminary results indicate that a significant number of operations related to inverse quantization can be still scheduled in the delay slots of the table lookup.

To make an absolute estimation of the performance we achieved, we mention that the maximum MPEG-2 compressed bit rate for Main Profile – Main Level (MP@ML) is 15 Mbit/s. For 16.9 cycle/coefficient, and an average of 5.4 bit/coefficient [7], the following rate can be processed in real-time by our implementation:

$$5.4\,\frac{\text{bit}}{\text{coefficient}} \;\times\; 200\cdot 10^{6}\,\frac{\text{cycle}}{\text{sec}} \;\times\; \frac{1}{16.9}\,\frac{\text{coefficient}}{\text{cycle}} \quad\approx\quad 64\,\frac{\text{M}bit}{\text{sec}}$$

That means that less than one-quarter of the computing power of the processor is used, or, equivalently, four MP@ML strings can be simultaneously (entropy) decoded.

5 Conclusions

We proposed a new entropy decoder implementation on TriMedia/CPU64 processor VLIW core which has the prologue exposed to the compiler. The VLC tables are organized in a special way such that an *end_of_block* or *error* will never be encountered during the prologue. By running preprocessed MPEG-2 conformance strings including only *run-level* and *end_of_block* symbols, we determined that the proposed entropy decoder is approximately 20% faster than its reference counterpart. In future work, we intend to evaluate the performance improvement for a complete MPEG decoder.

References

1. ***: MPEG-2 Video Codec. MPEG Software Simulation Group, WWW address: http://www.mpeg.org/MPEG/MSSG/
2. van Eijndhoven, J.T.J., Sijstermans, F.W., Vissers, K.A., Pol, E.J.D., Tromp, M.J.A., Struik, P., Bloks, R.H.J., van der Wolf, P., Pimentel, A.D., Vranken, H.P.E.: TriMedia CPU64 Architecture. In: IEEE Proceedings of International Conference on Computer Design (ICCD 1999), Austin, Texas (1999), 586–592.
3. ***: TM-1000 Data Book. Philips Electronics North America Corporation, TriMedia Product Group, Sunnyvale, California (1998).
4. Riemens, A.K., Vissers, K.A., Schutten, R.J., Sijstermans, F.W., Hekstra, G.J., Hei, G.D.L.: TriMedia CPU64 Application Domain and Benchmark Suite. In: IEEE Proceedings of International Conference on Computer Design (ICCD 1999), Austin, Texas (1999), 580–585.
5. Sun, M.T.: Design of High-Throughput Entropy Codec. In: VLSI Implementations for Image Communications. Volume 2. Elsevier Science Publishers B.V., Amsterdam, The Netherlands (1993), 345–364.
6. Mitchell, J.L., Pennebaker, W.B., Fogg, C.E., LeGall, D.J.: MPEG Video Compression Standard. Chapman & Hall, New York, New York (1996).
7. Pol, E.J.D.: VLD Performance on TriMedia/CPU64. Private Communication (2000).
8. Ishii, D., Ikekawa, M., Kuroda, I.: Parallel Variable Length Decoding with Inverse Quantization for Software MPEG-2 Decoders. In: Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS97), Leicester, United Kingdom (1997), 500–509.
9. Bonomini, F., Marco-Zompit, F.D., Milan, G., Odorico, A., Palumbo, D.: Implementing an MPEG2 Video Decoder Based on the TMS320C80 MVP. Application Report SPRA332, Texas Instruments, Paris, France (1996).
10. Haskell, B.G., Puri, A., Netravali, A.N.: Digital Video: An Introduction to MPEG-2. Kluwer Academic Publishers, Norwell, Massachusetts (1996).
11. Lei, S.M., Sun, M.T.: An Entropy Coding System for Digital HDTV Applications. In: IEEE Transactions on Circuits and Systems for Video Technology **1** (1991), 147–155.
12. Lin, H.D., Messerschmitt, D.G.: Finite State Machine has Unlimited Concurrency. In: IEEE Transactions on Circuits and Systems **38** (1991), 465–475.
13. Sima, M., Cotofana, S., Vassiliadis, S., van Eijndhoven, J.T.J., Vissers, K.: MPEG Macroblock Parsing and Pel Reconstruction on an FPGA-augmented TriMedia Processor. In: IEEE Proceedings of International Conference on Computer Design (ICCD 2001), Austin, Texas (2001), 425–430.
14. Johnson, W.M. *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, New Jersey (1991).
15. ***: Book 2 – Cookbook. Part D: Optimizing TriMedia Applications. TriMedia Technologies, Inc., TriMedia Technologies, Inc., Milpitas, California (2000).