MPEG-compliant Entropy Decoding on FPGA-augmented TriMedia/CPU64

Mihai Sima[†‡] Sorin Cotofana[†] Stamatis Vassiliadis[†] Jos T.J. van Eijndhoven[‡] Kees Vissers[§]
[†]*Delft University of Technology, Delft, The Netherlands*
[‡]*Philips Research, Eindhoven, The Netherlands*
[§]*TriMedia Technologies, Inc., Milpitas, California, U.S.A.*
E-mail: `M.Sima@et.tudelft.nl`

Abstract

The paper presents a Design Space Exploration (DSE) experiment which has been carried out in order to determine the optimum FPGA–based Variable-Length Decoder (VLD) computing resource and its associated instructions, with respect to an entropy decoding task which is to be executed on the FPGA-augmented TriMedia/CPU64 processor. We first outline the extension of the TriMedia/CPU64 architecture, which consists of an FPGA–based Reconfigurable Functional Unit (RFU) and the associated generic instructions. Then we address entropy decoding and propose a strategy to partially break the data dependency related to variable-length decoding. Three VLDs (VLD-1, VLD-2, VLD-3) instructions which can return 1, 2, or 3 symbols, respectively, are subsequently analyzed. After completing the DSE, we determined that VLD-2 instruction leads to the most efficient entropy decoding in terms of instruction cycles and FPGA area. The FPGA–based implementation of the computing resource associated to VLD-2 instruction is subsequently presented. When mapped on an ACEX EP1K100 FPGA from Altera, VLD-2 exhibits a latency of 8 TriMedia cycles, and uses all the Electronic Array Blocks and 51% of the logic cells of the device. The simulation results indicate that the VLD-2–based entropy decoder is 43% faster than its pure software counterpart.

1 Introduction

A common issue addressed by computer architects is the range of performance improvements that may be achieved by augmenting a general purpose processor with a reconfigurable core [1, 2, 3]. The basic idea of this approach is to take advantage of both the general purpose processor capability to achieve medium performance for a large class of applications, and FPGA flexibility to implement application-specific computations. An instance of such FPGA-extended processor is the TriMedia/CPU64+FPGA hybrid [4, 5], on which the user is given the freedom to define and use any computing facility subject to FPGA size and TriMedia/CPU64 organization. Several applications have been implemented on this hybrid. When the application exhibits data parallelism, a significant improvement over TriMedia/CPU64 alone have been achieved on FPGA-augmented TriMedia/CPU64 [4]. However, the improvement is rather low when data parallelism is not available, in particular for an entropy decoding task [5].

Entropy decoding consists of Variable-Length Decoding (VLD) [6, 7] followed by a Run-Length Decoding (RLD), from which the VLD is a strictly sequential task. Due to data dependency, the VLD is an intricate function on TriMedia, since a VLIW architecture must benefit from instruction level parallelism in order to be efficient. For this reason, VLD is an ideal candidate to benefit from reconfigurable hardware support. We have already proposed an FPGA–based MPEG-1 compliant VLD-1 computing resource and its associated instruction which returns one symbol [5].

In this paper, we demonstrate that significant improvements over the initial solution are possible with respect to entropy decoding, if a VLD instruction which can return more than one symbol is considered. The main idea in breaking the dependency between successive codewords is to determine the *run-level* pair, as well as the *code-length* for the *current* codeword, and only the *code-length* for the first, second, ..., *next* codewords during the same VLD call. Simultaneously, the *run-level* pairs for the incompletely decoded first, second, ..., *previous* codewords are computed. With the exception of a firing-up call, trully multiple-symbol decoding is achieved for all subsequent VLD calls.

After presenting the implementation of an MPEG-2 compliant VLD-1, VLD-2 and VLD-3 computing resources which can return two or three symbols, respectively, and their associated instructions are analized. The experimental results reveal that VLD-2 leads to the most efficient entropy decoding in terms of instruction cycles and FPGA area. When mapped on an ACEX EP1K100 FPGA from

Altera, VLD-2 computing resource exhibits a latency of 8 TriMedia cycles, and uses all the Electronic Array Blocks and 51% of the logic cells of the device. The simulations carried out on a TriMedia/CPU64 cycle accurate simulator indicate that VLD-2–based entropy decoder is 22% faster than our initial solution [5] at a slightly higher reconfigurable hardware cost, and 43% faster than the pure software solution [8]. Given the fact that TriMedia/CPU64 is a 5 issue-slot VLIW processor with 64-bit datapaths and a very rich multimedia instruction set, such an improvement within the target media processing domain indicates that the hybrid TriMedia/CPU64 + FPGA is a feasible approach with respect to entropy decoding.

Summarizing, the paper contributions are:

- A strategy to partially break the data dependency related to variable-length decoding.
- An FPGA–based MPEG-2–compliant VLD-1.
- An FPGA–based VLD-2 computing resource optimized with respect to the entropy decoding task, and in terms of instruction cycles and FPGA area.
- The syntax and the semantics of the VLD-2 custom operation.
- The VLD-2 implementation on Altera's ACEX EP1K100 FPGA.
- A high performance entropy decoder implementation on FPGA-augmented TriMedia/CPU64.

The paper is organized as follows. For background purposes, we outline several issues concerning MPEG compression standard, the architectures of the FPGA core, and the FPGA-augmented TriMedia/CPU64 in Section 2. MPEG-2–compliant VLD-1, VLD-2, VLD-3 implementations on FPGA are described in Section 3. VLD-x–based entropy decoders, where $x \geq 1$, are discussed in Section 4. The experimental framework and results are presented in Section 5. The final section completes the paper with some conclusions and closing remarks.

2 Background

The MPEG standard [7, 9] 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 for decreasing data storage requirements and data communication costs. A typical video codec system is presented in Figure 1 [6, 7]. 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; these are removed by the (lossless) entropy coder.

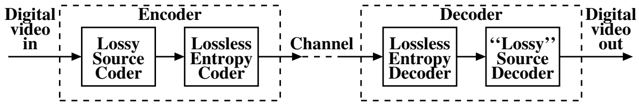

Figure 1. A generic video codec.

In MPEG, the couple DCT + Quantization is used as a lossy coding technique. The DCT algorithm processes 8×8 blocks of pixels, and outputs 8×8 blocks of coefficients representing the amplitudes of 64 spatial frequencies. Since the human eye cannot readily perceive high spatial frequency activity, a quantization step is then carried out. As a result, a lot of elements of the 8×8 matrix become zero. 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, also referred to as *source 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 source symbols and *variable length codewords* 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 can be 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 assigned to separate between two 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 cannot be pipelined.

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 with 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. Generally speaking, a VLD contains a look-up table which receives the variable-length code itself as the address [10], as depicted in Figure 2. The decoded symbol (*run/level* pair or *end-of-block*) and the codeword length are generated in response to that address. In order to determine the starting position of the next codeword, the code-length is added to the previous code-length sum. Since the longest codeword excluding Escape has 17 bits, the LUT size reaches $= 2^{17}$ words for a direct mapping of all possible codewords.

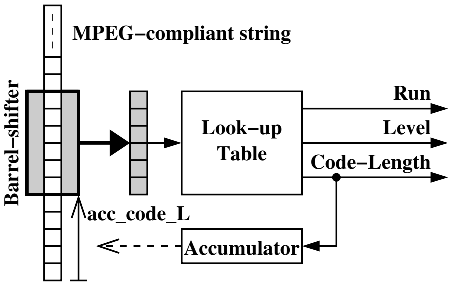

Figure 2. Variable-length decoding principle.

Conceptually, for each *run/level* pair returned by the VLD, the RLD 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 at positions defined by *run* values, as depicted in Figure 3. This common strategy has been widely used in previous work [11, 12, 5] and will be used subsequently, too.

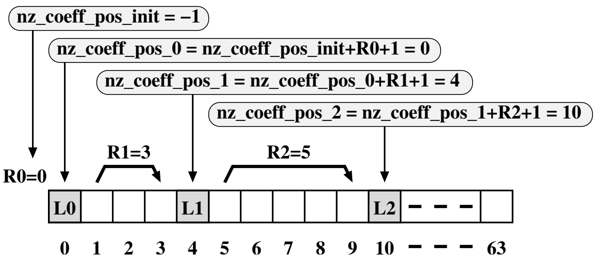

Figure 3. Run-length decoding principle.

In the figure, the position of a non-zero coefficient, `nz_coeff_pos`, is computed by adding the *run* value, R, together with an '1' bit to the position of the previous non-zero coefficient.

The next subsection will outline the architecture of the FPGA we used as an experimental reconfigurable core.

2.2 The FPGA architecture

Field-Programmable Gate Arrays (FPGA) [13] are devices which can be configured *in the field* by the end user. In a general view, an FPGA is composed of two constituents: *Raw Hardware* and *Configuration Memory*. The function performed by the raw hardware is defined by the information stored into the configuration memory. In the sequel, we will assume that the architecture of the raw hardware is identical to that of an ACEX 1K device from Altera [14]. Briefly, an ACEX 1K device contains an array of Logic Cells, each including a 4-input Look-Up Table, a number of Embedded Array Blocks (EAB), each EAB being actually a RAM block with 8 inputs and 16 outputs, and an interconnection network. In order to have a general view, we mention that the logic capacity of the ACEX 1K family ranges from 576 logic cells and 3 EABs for an EP1K10 device to 4992 logic cells and 12 EABs for an EP1K100 device. More details regarding the architecture and operating modes of ACEX 1K devices can be found in [14].

We conclude this section with a review on the architectural extension for the TriMedia/CPU64.

2.3 Architectural extension for TriMedia/CPU64

TriMedia/CPU64 is a 64-bit 5 issue-slot VLIW core, launching a long instruction every clock cycle [15]. 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. The architecture supports subword parallelism and is optimized with respect to media-processing. For example, operations on eight 8-bit unsigned integers (`vec64ub`), or on four 16-bit signed integers (`vec64sh`) are possible. The TriMedia/CPU64 VLIW core also supports double-slot operations, or super-operations. Such a super-operation occupies two adjacent slots in the VLIW instruction, and maps to a double-width functional unit. This way, operations with more than 2 arguments and one result are possible.

As described in [4], TriMedia/CPU64 processor can be augmented with a Reconfigurable Functional Unit (RFU) which consists mainly of an FPGA core. The RFU is embedded into the TriMedia as any other hardwired functional unit, i.e., it receives instructions from the instruction decoder, reads its input arguments from and writes the computed values back to the register file. In order to use the RFU, new generic instructions are provided [4]: `SET`, and `EXECUTE`. Loading a new configuration into RFU is performed under the command of a `SET` instruction, while `EXECUTE` instructions launch the operations performed by the computing resources which are already configured on the raw hardware. Thus, the execution of an RFU-mapped operation requires two basic stages: `SET`, and `EXECUTE`.

In the next section, three FPGA–based variable-length decoders which can return 1, 2, respectively 3 symbols per call are described.

Table 1. The partitioning of the VLC codes (Table B14) into groups and classes [5].

Name of the group	No. of symbols in the class	Class prefix	Code length	Group header	Effective EAB address length
NI-1st coeff. Group 0	2	1	1 + s	–	n.a.
End-of-block	1	10	2	–	n.a.
NI-subsequent/I-AC coeff. Group 0	2	11	2 + s	–	n.a.
Escape MPEG-2/(MPEG-1)	1	0000 01	6 + 18/(14,22)	–	n.a.
	2	011	3 + s		3
	4	010	4 + s		4
Group 1	4	0011	5 + s		5
	2	0010 1	5 + s	0	5
(implemented	8	0001	6 + s		6
into EABs)	8	0000 1	7 + s		7
	16	0010 0	8 + s		8
Group 2	16	0000 001	10 + s		5
(implemented	32	0000 0001	12 + s	0000 00	7
into EABs)	32	0000 0000 1	13 + s		8
Group 3	32	0000 0000 01	14 + s		6
(implemented	32	0000 0000 001	15 + s	0000 0000 0	7
into EABs)	32	0000 0000 0001	16 + s		8

3 Variable-length decoders

Due to data dependency, the VLD is an intricate function on TriMedia, since a VLIW architecture must benefit from instruction level parallelism in order to be efficient. For this reason, VLD is an ideal candidate to benefit from reconfigurable hardware support. In this section, we will first present an MPEG-2 compliant VLD which can return one symbol per call. Then, we will propose a strategy to break the data dependency between successive symbols, and propose VLD-2 and VLD-3 computing resources which can return 2, respectively 3 symbols per call.

3.1 VLD-1

VLD-1 is an FPGA–based VLD which can decode one symbol per execution. Since the latency of an RFU-configured computing resource should be known at compile time, only a constant-output-rate architecture [10, 16, 6] can be considered for VLD-1. An FPGA–based VLD-1 which is only MPEG-1 compliant (i.e., only Tables B12, B13, B14 [17] are implemented) has been presented briefly in [5]. The main idea of this design is to compute the *run-level* pair by looking-up into the (8-input) EABs of an ACEX 1K FPGA, while the *code-length* and control information are computed into FPGA logic cells. For this reason, the B14 table has been partitioned into Groups and Classes, each Class being defined by a *prefix* (Table 1). By bypassing the *header* which is common to all codewords in the Group, the number of remaining bits to be decoded, and, therefore, the *effective address length* for EAB, for each and every codeword is 8 or less. Six EABs are needed to implement the Groups 1, 2, and 3.

In order to cover all instances of the MPEG-2 standard [17] regarding DCT coefficient decoding, we further propose a strategy to implement the B15 table. The partitioning of the B15 table in groups and classes is presented in Table 2. The slightly higher difficulty of having a significant number of codewords starting with a '1' has been solved as follows: the codewords starting with "10" or "01" have been allocated to Group 0, while the codewords starting with "00" or "11" have been allocated to Group 1. In this way, each class in the Group 1 can be uniquely identified by the second most-significant bit. Therefore, the first most-significant bit, which represent the *header* of the Group 1, can be bypassed as in the Table B14 case.

A combinatorial circuit has been configured on FPGA which can compute the code-length of the symbol. By means of EABs, the *run* and *level* for each and every group were decoded in parallel, as the valid symbol would belong to that group. Then, a selection of the proper run and level pair is carried out according to the code-length, as depicted in Figure 4 (The first bit of the VLC string is labeled as bit No. 0.). The Figure presents only the VLD-1 core, which means that specifying the decoding parameters *mpeg-2/mpeg-1*, *intra/non-intra*, *intra-vlc-format*, *luma/chroma*, *dc/ac*, is left open for optimizations at the entropy decoder level.

It should be mentioned that all the groups 1, 2, and 3 of both tables B14 and B15 fit into a single ACEX EP1K100 FPGA. All 12 EABs and 22% of the logic cells of an ACEX EP1K100 device have been used to implement the MPEG-

Table 2. The partitioning of the VLC codes (Table B15) into groups and classes.

Name of the group	No. of symbols in the class	Class prefix	Code length	Group header	Effective EAB address length
End-of-block	1	0110	4	–	n.a.
Group 0	2	10	2 + s	–	n.a.
	2	010	3 + s	–	n.a.
	2	0111	4 + s	–	n.a.
Escape MPEG-2	1	0000 01	6 + 18	–	n.a.
Group 1	4	0011	5 + s	0	5
	2	0010 1	5 + s		5
	8	0001	6 + s		6
	8	0000 1	7 + s		7
	16	0010 0	8 + s		8
(implemented into EABs)	2	110	3 + s	1	3
	4	1110	5 + s		5
	8	1111 0	7 + s		7
	2	1111 100	7 + s		7
	8	1111 11	8 + s		8
	4	1111 101	8 + s		8
Group 2 (implemented into EABs)	2	0000 0010	9 + s	0000 00	4
	4	0000 0011 1	9 + s		4
	4	0000 0011 0	10 + s		5
	20	0000 0001	12 + s		7
	24	0000 0000 1	13 + s		8
Group 3 (implemented into EABs)	32	0000 0000 01	14 + s	0000 0000 0	6
	32	0000 0000 001	15 + s		7
	32	0000 0000 0001	16 + s		8

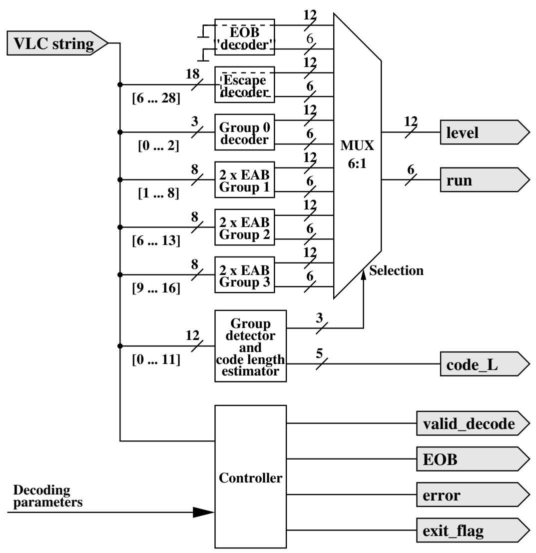

Figure 4. The VLD-1 core on FPGA – from [5].

2–compliant VLD-1. By simulation with Altera tools, we found that the VLD-1 latency is equal to 7 TriMedia cycles.

Since the next variable-length codeword can be decoded only after the current one has been decoded, VLD is a strictly sequential task. Subsequently, we will propose a strategy to increase the parallelism by breaking the explicit dependency between successive codewords, and will present a variable-length decoder which can decode 2 symbols per call.

3.2 VLD-2

Since VLD is a strictly sequential algorithm, trully *two codewords at a time* can be achieved only if a huge look-up table of $28 + 28 = 56$ inputs (2^{56} word memory!) for MPEG-1, and $24 + 24 = 48$ inputs (2^{48} word memory!) for MPEG-2 is employed. Even by excluding the ESCAPE codes, a look-up memory with $17 + 17 = 34$ inputs, which means $2^{34} = 16$ Gwords is still needed! Such a large memory is impractical for the time being.

Several VLD architectures, which return two symbols per execution, have been proposed. We will summarize them subsequently:

- The main idea of the architecture proposed by Park [18] is to find the *run*, *level*, and *code-length* of a first symbol, then barrel-shift the VLC string according to the code-length, and finally find the *run*, *level*, and *code-length* of a subsequent symbol. Since a barrel-shifter can be implemented on an ACEX 1K FPGA only by means of cascaded multiplexers selecting fixed-size shifting by $1, 2, 4, \ldots$, respectively, such approach exhibits high latency and large raw hardware utilization. Consequently, this architecture is not appropriate for our case. However, if full-custom (ASIC) implementation is addressed, such architectural solution might be reconsidered.

- The architectures described in [19], [20] employ *advance computation* techniques. For the first (most significant) bit of the VLC string, the symbol is fully decoded, i.e., *run*, *level*, and *code-length* are determined. In parallel, for all possible starting bit positions for the next symbol, *run*s, *level*s, *code-length*s are also generated. Finally, only a selection based on the codeword length of the first decoded symbol is carried out. The major drawback of this approach is the huge complexity of the decoder, as decoding look-up tables have to be provided for the first symbol, and also for all possible second symbols.

In order to overcome the drawbacks of the above mentioned architectures, we propose to decode *run*, *level*, and *code-length* of a first symbol, and to determine only the code-length for the second symbol by means of advance computation techniques. The computation of the *run-level* pair of the second symbol is postponed to the next VLD call. In parallel, the *run-level* pair of the *previous* codeword is determined. The complexity of the VLD-2 remains resonable low since only a small number of decoding look-up tables have to be provided. The barrel-shifting is intended to be carried out in software, by the TriMedia/CPU64 core.

Several aspects regarding the terminology have to be discussed. The codeword corresponding to the first (most significant) bit of the VLC string will be referred to as *current* codeword. The second symbol will be referred to as *next* codeword, and a codeword whose code-length was determined during the previous call of VLD-2 will be referred to as *previous* codeword. The acronyms related to the *current* codeword get the suffix `_c`, those related to *next* codeword get the suffix `_n`, and those related to the *previous* codeword get the suffix `_p`. Therefore, acronyms like `run_c`, `level_c`, `code_L_c`, `code_L_n`, `run_p`, `level_p`, etc, are considered as valid.

The FPGA–based implementation of the VLD-2 core is presented in Figure 5. The same methodology used in VLD_1 has been employed, i.e., *run* and *level* for all the Groups in the decoding tables are decoded in parallel, then only a selection of the proper result is carried out. In order to easily quit the entropy decoder calling routine once an *end-of-block* or an error has been detected for either of the *current* or *previous* codeword, a global exit flag has been provided.

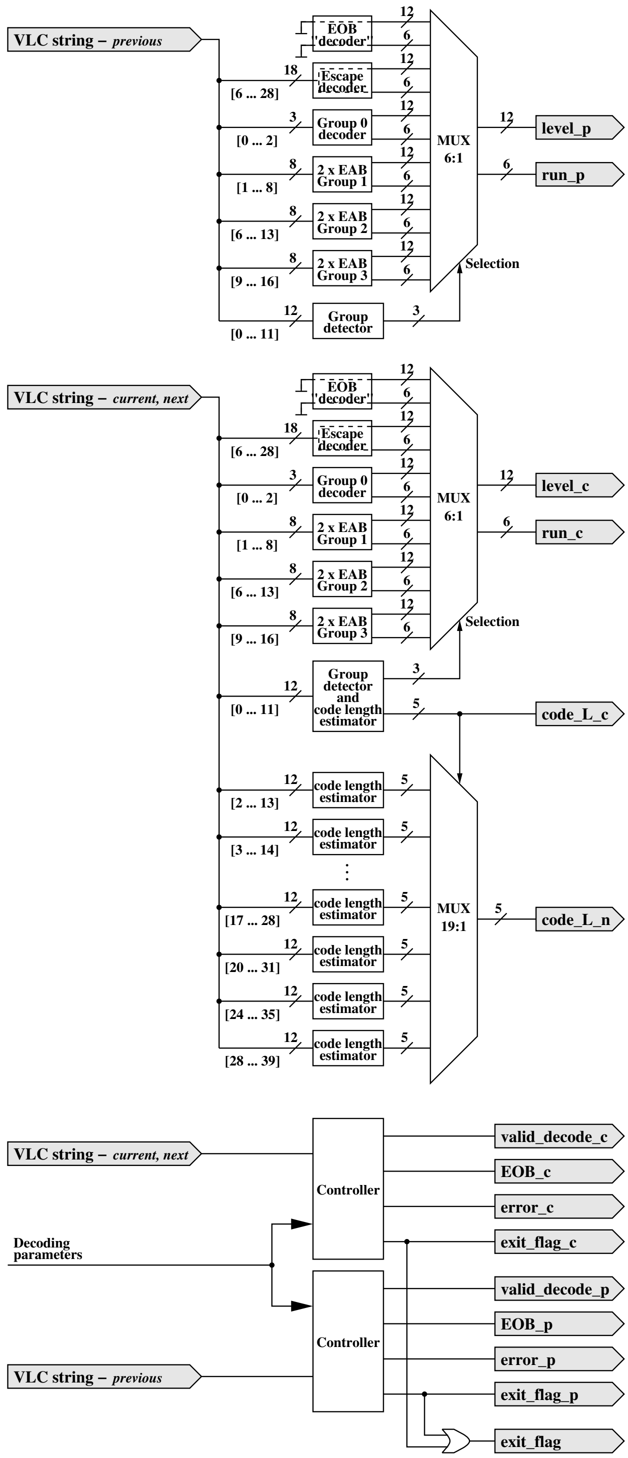

Figure 5. The VLD-2 core on FPGA.

All 12 EABs and 51% of the logic cells of an ACEX EP1K100 device have been used to implement VLD-2 embedding either B14 or B15 table. To implement both tables, either two EP1K100 devices are needed, or a multiple-context [21] EP1K100 should be available. Since there is no need to simultaneously decode according to both B14 and B15 tables, the multiple-context solution is preferred. By simulation with Altera tools, we found that the VLD-2 latency ranges between 7-8 TriMedia cycles, depending on the computing of additional values which may prove useful at the entropy decoding routine level, e.g., `nz_coeff_pos`, `end-of-macro-block`, etc.

3.3 VLD-3

The VLD-2 principle is scalable and can be extended to VLD-x ($x \geq 3$), subject to the FPGA size. In a VLD-3, two *next/previous* codewords are considered. Unfortunately, VLD-x ($x \geq 3$) seems not to be feasible. In VLD-2, the selection of the proper `code_L_n` can be completed in about the same time with *run-level* decoding for the *current* and *previous* codewords. In VLD-3, for example, the computation of the code-lengths for the *next* two codewords is on the critical path. Therefore, longer latencies are to be expected for VLD-x ($x \geq 3$), while VLD-2 has about the same latency with VLD-1. There are also limitations connected with TriMedia/CPU64 super-operation format, which strongly discourages using VLD-x ($x \geq 3$), as we will describe later, in Section 5.

In entropy decoding on FPGA-augmented TriMedia, the VLD benefits from reconfigurable hardware support, while the processor still carries out the inverse zig-zag and matrix reconstruction, i.e., the RLD. In Section 4, we will describe three entropy decoders on FPGA-augmented TriMedia.

4 Entropy decoders

Since the datapath width of TriMedia/CPU64 is 64 bit, the MPEG-compliant string is downloaded into the TriMedia core in chunks of 64 bits. The *accumulated code length* variable, `acc_code_L`, represents the sum of the code-lengths of the already decoded codewords. Before each VLD call, the MPEG string has to be shifted `acc_code_L` positions in order to discard these codewords. After the VLD returns, `acc_code_L` is incremented modulo-64 with the code-length of the newly decoded codeword(s). This way, a new chunk of 64 bits from the MPEG string is downloaded into TriMedia core on overflow.

4.1 VLD-1–based entropy decoder

The main idea in an FPGA-based solution is to replace the awkward table look-ups of the pure software decoder [12], by a single RFU call. In this subsection, we will consider that a VLD computing resource which can decode one symbol per execution (call) – the VLD-1 – is to be implemented on FPGA. Thus, the entropy decoder includes calls to VLD-1. The following stages can be discerned in the implementation of the VLD-1–based entropy decoder:

1. **Initializations**.
2. **VLD-1 call**
3. **Field extraction**: `run`, `level`, `code_L`, `exit_flag`.
4. **Updating the accumulated code-length**:

   $\texttt{acc\_code\_L} += \texttt{code\_L}$ (modulo 64)
5. **Exit** if an exit condition (*end-of-block*, error, etc.) has been encountered.
6. **Run-length decoding** (updating `nz_coeff_pos` followed up by filling-in the non-zero coefficient into the 8×8 matrix) **and additional computations associated with MPEG** (de-zig-zag, inverse quantization, etc.).
7. **Aligning the VLC string** in order to bypass the already decoded bits.
8. **Looping: GOTO step 2**.

The Stage 6 – **run-length decoding** – can be folded into the loop, such that loop pipelining is employed. In this way, the run-level decoding for the *previous* decoded symbol is carried out simultaneously with variable-length decoding of the *current* symbol.

The VLD-2–based entropy decoder is a direct extension of the VLD-1–based one. We will describe it subsequently.

4.2 VLD-2–based entropy decoder

The entropy decoder routine includes calls to VLD-2. The following stages can be discerned in the implementation of the VLD-2–based entropy decoder:

1. **Initializations**.
2. **VLD-2 call**
3. **Field extraction**: `run_p`, `level_p`, `code_L_n`, `run_c`, `level_c`, `code_L_c`, `exit_flag`.
4. **Updating the accumulated code-length**:

   $\texttt{acc\_code\_L} += \texttt{code\_L\_c}$ (modulo 64)
5. **Aligning of the VLC string** in order to compute *VLC string - previous*.
6. **Updating the accumulated code-length**:

   $\texttt{acc\_code\_L} += \texttt{code\_L\_n}$ (modulo 64)
7. **Exit** if an exit condition (*end-of-block*, error, etc.) has been encountered.
8. **Run-length decoding** for *previous* symbol (updating `nz_coeff_pos_p` followed up by filling-in the non-zero coefficient into the 8×8 matrix) **and additional computations associated with MPEG** (de-zig-zag, inverse quantization, etc.).

9. **Run-length decoding** for *current* symbol (updating `nz_coeff_pos_c` followed up by filling-in the non-zero coefficient into the 8×8 matrix) **and additional computations associated with MPEG** (de-zig-zag, inverse quantization, etc.).

10. **Aligning of the VLC string** in order to compute *VLC string - current, next.*

11. **Looping: GOTO step 2**.

Following the same strategy described in Subsection 4.1, either the Stage 9 or both Stages 9 and 8 can be folded into the loop. It can be easily observed that the complexity of the entropy decoding loop is definitely higher than that of its VLD-1–based counterpart. For this reason, the overhead associated with firing-up the loop may become significant and even cancel the efficiency provided by VLD-2. Which folding strategy leads to shorter decoding time will be determined by experiment.

4.3 VLD-3–based entropy decoder

VLD-2–based entropy decoder can be extended to an VLD-x–based ($x \geq 3$) entropy decoder. Unfortunately, issues related with TriMedia super-operation format limit the utilization of a VLD-x ($x \geq 3$) computing resource. Also, the overhead associated with firing-up the loop becomes larger and larger, which turns into a highly inefficient VLD-x–based ($x \geq 3$) entropy decoder. We will come back to these issues and present more details as well as experimental figures regarding VLD-3–based entropy decoder in Section 5.

With VLD-1, VLD-2, and VLD-3, different tests have been carried out. We will present them subsequently along with experimental results.

5 Experimental results

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 matries 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 the instruction cycles required to perform strictly entropy decoding. Therefore, the main goal was to minimize this number.

Two experiment classes have been considered. In the first class, VLD-1, VLD-2 and VLD-3 generate an *exit* condition if an *end-of-block* has been encountered. This way, the entropy decoding loop is left after the 8×8 block has been fully reconstructed. As mentioned, only a single codeword is decoded on the first VLD-x ($x \geq 2$) call. Since the average number of codewords per block is quite small, ranging between 5 and 6 for non-intra macroblocks [12], the inefficiency of the first VLD-x ($x \geq 2$) is significant. For example, 3 VLD-2 calls instead of the ideal 2.5 are needed to decode 5 coefficients (20% “overhead”), while 4 VLD-2 calls instead of the ideal 3 are needed to decode 6 coefficients (25% “overhead”).

In the second experiment class, VLD-2 and VLD-3 generate an *exit* condition if an *end-of-macro-block* has been encountered, and the entropy decoding loop is left only after the entire macroblock has been reconstructed. There are about 15 codewords per block for non-intra macroblocks [12]. Considering again VLD-2, 8 instead of the ideal 7.5 VLD-2 calls are needed to decode a macroblock. Thus, the first iteration “overhead” is much lower (only 7%). We have to note that there is no *end-of-macro-block* symbol. An *end-of-macro-block* condition is rised if the *end-of-block* is encountered when the last block of the macroblock is being decoded.

For both experiment classes, the performances of the entropy decoders have been evaluated according to two scenarios. In the first scenario, the VLDs return the `run` value as defined by the MPEG standard, while the position of the non-zero coefficient, `nz_coeff_pos`, in the (macro)block is returned in the second scenario. When the `nz_coeff_pos` represent the position in a block, the block index, `block_index`, in the macroblock is also returned in the EOMB experiment class.

In the sequel, we will present the VLD-x associated instructions. The sintax of the VLD-1 and VLD-2 instructions are quite similar:

$$\texttt{VLD\_1 Ry} \rightarrow \texttt{Rz, Rw}$$

$$\texttt{VLD\_2 Ry, Ryy} \rightarrow \texttt{Rz, Rw}$$

The registers Ry and Ryy contain the incoming coded string which has been aligned to start with the *current*, respectively *previous* codeword. The *run* (or alternatively, *nz_coeff_pos*), and the *level* for both *current* and *previous* codeword are each represented on a 16-bit signed integer, and stored together as a four 16-bit signed integer vector in the Rz register. Even though *run* (or *nz_coeff_pos*) is always a positive number which can be represented on 6 bits (10 bits for *nz_coeff_pos*), our solution is more effective with

Table 3. Entropy decoding experimental results.

Entropy Decoder MPEG benchmark	Block type	Workload (coeff.)	**Pure software** (cycles)	VLD-1–based (cycles)	**VLD-2–based** (cycles)	VLD-3–based (cycles)	**Improvement**
bat_327_334	I (B15)	172,745	**2,843,376**	2,050,693	**1,618,656**	1,799,032	**44.2 %**
	NI	266,485	**4,592,358**	3,112,249	**2,534,072**	2,768,638	
popplen	I (B15)	47,003	**777,553**	546,243	**435,114**	474,281	**43.3 %**
	NI	28,069	**475,326**	379,466	**275,753**	301,552	
sarnoff2	I (B14)	80,563	**1,387,489**	946,538	**748,844**	822,253	**42.8 %**
	NI	36,408	**577,388**	485,585	**375,558**	412,659	
tennis	I (B14)	12,345	**210,011**	149,366	**118,943**	131,284	**43.4 %**
	I (B15)	120,754	**1,937,808**	1,421,498	**1,109,466**	1,248,815	
	NI	137,756	**2,527,395**	1,795,628	**1,416,234**	1,597,229	
ti1cheer	I (B15)	80,818	**1,311,687**	970,904	**749,386**	823,350	**41.3 %**
	NI	51,680	**836,082**	667,417	**512,389**	573,799	

respect to splitting the Rz vector into its components. Indeed, a single TriMedia/CPU64 cycle is needed to extract an element from a vector. The register Rw is an eight 8-bit unsigned integer vector and contains the *code-length*s of the *current* and *next* codewords, *block index*-es associated to *current* and *previous* codewords in the EOMB class, as well as control information for each and every *previous*, *current*, and *next* codewords. We decided to provide for redundant control information such as *error*, *valid_decode*, and *EOB* flags, in order to help the entropy decoder's calling routine to deal with error concealment [17, 7, 9]. A `global_exit` flag which is set up when any exit condition is raised is also provided.

The same strategy to pack *run*/*nz_coeff_pos* and *level* values into a four 16-bit signed integer vector is no longer possible in the VLD-3. Since there are too many values which have to be returned by the VLD-3 call (three *run*s/*nz_coeff_pos*, three *level*s, three *code-length*s, three *block_index*es for the EOMB class, and the control information), the only possible solution to pack them in a 64-bit word is to cross the boundaries between bytes. Thus, field extraction will be performed by a sequence of mask, shift and OR operations. Consequently, at least three TriMedia/CPU64 cycles instead of a single one will be needed.

The reference for evaluating the performance of FPGA–based VLDs is a pure software entropy decoder [8], which is itself an improved version of the one proposed by Pol [12]. Since the pure software implementation is out of the paper scope, we will not go into details. However, we still mention that by running our pure software entropy decoder on a TriMedia/CPU64 cycle accurate simulator over a set of MPEG conformance bitstreams, we determined that 4 of 5 issue slots are filled in with operations (by comparison, we mention that 2.9 of 5 issue slots are filled in with operations in Pol's implementation). This result which updates our initial assumption about the efficiency of the pure software implementation [5] is indeed a challenging reference for TriMedia/CPU64+FPGA hybrid.

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. All pure software, VLD-1–based, VLD-2–based, and VLD-3–based entropy decoders were run on the TriMedia/CPU64 cycle accurate simulator over each of the modified MPEG string. The best results for each entropy decoder are presented in Table 3. The figures indicate the number of instruction cycles needed to decode the pre-processed MPEG string. The relative improvement specified in the last column of the Table has been computed with reference to our pure software entropy decoder.

Comparing the figures of the EOB experiment class with the figures of EOMB class is a little unfair from the EOMB point of view, for more functionality is considered in the later class. Since the entropy decoder delivers the entire macroblock on completion in the EOMB class, and only an 8×8 block in EOB class, block reconstruction is not carried out in EOB experiment class. Since this extra functionality for managing macroblock reconstruction is subject to optimizations at a complete MPEG decoder level, this is the best we can do for the time being. Therefore, we proceed to a conservative evaluation, accept this unfair comparison, and claim that the FPGA-augmented TriMedia/CPU64 can perform entropy decoding 43% faster than the standard TriMedia/CPU64. Given the fact that TriMedia/CPU64 is a 5 issue-slot VLIW processor with 64-bit datapaths and a very rich multimedia instruction set, such an improvement within the target media processing domain indicates that the hybrid TriMedia/CPU64 + FPGA is a feasible approach for entropy decoding.

6 Conclusions. Future Work

We proposed a strategy to partially break the data dependency related to variable-length decoding. An FPGA–based VLD-2 computing resource optimized in respect with the entropy decoding task has been described. All electronic-array blocks and 51% of the logic cells of an ACEX EP1K100 FPGA have been used to implement VLD-2. A VLD-2–based entropy decoder running on FPGA-augmented TriMedia/CPU64 is 43% more efficient than its pure software counterpart. In future work, we intend to evaluate the performance improvement for a complete MPEG decoder.

References

[1] R. Razdan and M.D. Smith, "A High Performance Microarchitecture with Hardware-Programmable Functional Units," in *27th Annual International Symposium on Microarchitecture – MICRO-27*, San Jose, California, November 1994, pp. 172–180.

[2] R.D. Wittig and P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, April 1996, pp. 126–135.

[3] J.R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, April 1997, pp. 12–21.

[4] M. Sima, S.D. Cotofana, J.T.J. van Eijndhoven, S. Vassiliadis, and K.A. Vissers, "8×8 IDCT Implementation on an FPGA-augmented TriMedia," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Rohnert Park, California, April 2001.

[5] M. Sima, S.D. Cotofana, S. Vassiliadis, J.T.J. van Eijndhoven, and K.A. Vissers, "MPEG Macroblock Parsing and Pel Reconstruction on an FPGA-augmented TriMedia Processor," in *IEEE International Conference on Computer Design*, Austin, Texas, September 2001, pp. 425–430.

[6] M.-T. Sun, *VLSI Implementations for Image Communications*, vol. 2, chapter Design of High-Throughput Entropy Codec, pp. 345–364, Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1993.

[7] J.L. Mitchell, W.B. Pennebaker, C.E. Fogg, and D.J. LeGall, *MPEG Video Compression Standard*, Chapman & Hall, New York, New York, 1996.

[8] M. Sima, "MPEG-compliant Entropy Decoding on TriMedia/CPU64," Private Communication, December 2001.

[9] B.G. Haskell, A. Puri, and A.N. Netravali, *Digital Video: An Introduction to MPEG-2*, Kluwer Academic Publishers, Norwell, Massachusetts, 1996.

[10] S.-M. Lei and M.-T. Sun, "An Entropy Coding System for Digital HDTV Applications," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 1, no. 1, pp. 147–155, March 1991.

[11] MPEG Software Simulation Group, "MPEG-2 Video Codec," http://www.mpeg.org/MPEG/MSSG/

[12] E.-J. Pol, "VLD Performance on TriMedia/CPU64," Private Communication, May 2000.

[13] S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A Tutorial," *IEEE Transactions on Design and Test of Computers*, vol. 13, no. 2, pp. 42–57, 1996.

[14] Altera Corporation, "ACEX 1K Programmable Logic Family," Datasheet, San Jose, California, April 2000.

[15] J.T.J. van Eijndhoven, F.W. Sijstermans, K.A. Vissers, E.-J. Pol, M.J.A. Tromp, P. Struik, R.H.J. Bloks, P. van der Wolf, A.D. Pimentel, and H.P.E. Vranken, "TriMedia CPU64 Architecture," in *Proceedings of International Conference on Computer Design*, Austin, Texas, October 1999, pp. 586–592.

[16] M.-T. Sun and K.-H. Tzou, "High-Speed Flexible Variable-Length-Code Decoder," U.S. Patent No. 5,173,695, December 1992.

[17] International Telecommunication Unit, "Information technology – Generic coding of moving pictures and associated audio information: Video," ITU-T Recommendation H.262, February 2000.

[18] Y.-G. Park, "High Speed Variable Length Code Decoding Apparatus," U.S. Patent No. 5,561,690, October 1996.

[19] H.-D. Lin and D.G. Messerschmitt, "Designing a High-Throughput VLC Decoder. Part II – Parallel Decoding Methods," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 2, no. 2, pp. 197–206, June 1992.

[20] S. Kinouchi and A. Sawada, "Huffman Code Decoding Circuit," U.S. Patent No. 5,617,089, April 1997.

[21] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, April 1994.