Variable Length Decoder Implemented on a TriMedia/CPU64 Reconfigurable Functional Unit

Mihai Sima[†‡] Sorin Cotofana[†] Stamatis Vassiliadis[†] Jos T.J. van Eijndhoven[‡]
[†]*Delft University of Technology, – Dept. of Electrical Engineering, Delft, The Netherlands*
[‡]*Philips Research, – Dept. of Information and Software Technology, Eindhoven, The Netherlands*
Phone: +31-(0)40-274-2593 E-mail: M.Sima@et.tudelft.nl

Abstract— **This paper presents the implementation of a Variable Length Decoder (VLD) on a TriMedia/CPU64 (FPGA-based) Reconfigurable Functional Unit (RFU). The VLD decodes run-level symbols coded as Variable-Length Codes (VLC) following the MPEG standard. The RFU consists mainly of an Altera FPGA core, and 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. Variable-length decoding is an awkward computational part of MPEG decoding. Since the location of a symbol in the stream depends of the data which precedes it, every symbol has to be decoded before the following one can be. In order to balance the complexity and the efficiency of the RFU-mapped decoder, we propose a VLD computing facility which returns a single VLC symbol (*run/level* pair or *end-of-block*) per operation. Briefly, the VLD is implemented as a parallel lookup into FPGA's Electronic Array Blocks (EAB), followed by a selection of the proper result. Since each EAB can implement a lookup table of 8 inputs, the VLC symbols have been partitioned into classes according to this FPGA architectural characteristic. When mapped on an ACEX EP1K100 FPGA, our VLD computing resource exhibits a latency of 7 TriMedia cycles, and uses 6 EABs and 16% of the logic cells of the device.**

Keywords— **Reconfigurable computing, variable-length decoder, VLIW processors, field-programmable gate array.**

I. 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. The basic idea of such approach is to exploit both the general purpose processor capability to achieve medium performance for a large class of applications, and FPGA flexibility to implement application-specific computations.

This paper presents the implementation of a Variable-Length Decoder (VLD) on a TriMedia/CPU64 FPGA-based Reconfigurable Functional Unit (RFU). TriMedia/CPU64 is a 64-bit 5 issue-slot VLIW processor launching a long instruction every clock cycle. The RFU consists mainly of an FPGA core, and 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. With such RFU, the user is given the freedom to define and use any computing facility subject to the FPGA size and TriMedia organization.

Variable-length decoding is an awkward computational part of MPEG decoding. Since the location of a VLC symbol in the stream depends of the data which precedes it, every symbol has to be decoded before the following one can be. In other words, the entire MPEG stream must be decoded serially. In order to balance the complexity and the efficiency of the RFU-mapped decoder, we propose a VLD computing facility which returns a single VLC symbol (*run/level* pair or *end-of-block*) per operation. Briefly, the VLD is implemented as a parallel lookup into FPGA's Electronic Array Blocks (EAB), followed by a selection of the proper result. Since each EAB can implement a lookup table of 8 inputs, the VLC symbols have been partitioned into classes according to this FPGA architectural characteristic. When mapped on an ACEX EP1K100 FPGA, our VLD computing resource exhibits a latency of 7 cycles, and uses 6 EABs and 16% of the logic cells of the device.

The paper is organized as follows. For background purpose, we briefly present the most important issues related to VLD theory and architecture of the reconfigurable core in Section II. Section III outlines the architectural extension of TriMedia/CPU64. Implementation issues of the VLD computing resource on FPGA are presented in Section IV. Section V completes the paper with some conclusions and closing remarks.

II. Background

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 [1], [2]. The lossy source coder performs filtering, transformation (such as Discrete Cosine Transform (DCT), subband decomposi-

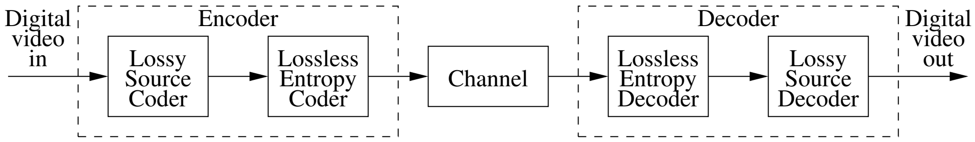

Fig. 1. The block diagram of a generic video codec – adapted from [1], [2].

tion, or differential pulse-code modulation), quantization, etc. The output of the source coder still exhibits various kinds of statistical dependencies. The (lossless) entropy coder exploits the statistical properties of data and removes the remaining redundancy after the lossy coding.

In MPEG, the DCT is used as a lossy coding technique. The DCT algorithm processes the video data in blocks of 8×8, decomposing each block into a weighted sum of 64 spatial frequencies. At the output of DCT, the data is also organized in 8×8 blocks of coefficients, each coefficient representing the contribution of a spatial frequency for the block being analyzed. Following a zig-zag path, the matrix is transformed into a vector of coefficients, and further compressed by an entropy coder which consists of two major parts: Run-Length Coder (RLC) and 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 pairs of *zero-run lengths* and *values* of quantized DCT coefficients. When all the remaining coefficients in a vector are zero, they are all coded by the special symbol *end-of-block*. Variable length coding, also known as Huffman coding, is a mapping process between source symbols and *variable length codewords*. The variable length coder assigns shorter codewords to frequently occuring source symbols, and vice versa, so that the average bit rate is reduced. In order to achieve maximum compression, the coded data is sent through a continuous stream of bits with no specific guard bit assigned to separate between two consecutive symbols. As a result, decoding procedure must recognize the code length as well as the symbol itself.

Subsequently, we will briefly present the theoretical background of variable-length decoding.

A. Variable-Length Decoder

The input to VLD is the encoded bit stream, and the output is the decoded symbols. Since the code length of the symbol is variable, both the input and output bit rate of the decoder cannot be kept constant. As described in [1], three different decoder types are possible: constant input rate, constant output rate, and variable input-output rate.

The *constant-input-rate* VLD decodes a fixed number of bits and produces a variable number of symbols per unit time. An example of such decoder which decodes one bit per cycle is described in [3]. The decoder employs a binary tree search technique in which a token is propagated in a reverse Huffman tree constructed from the original codes. Although some improvements of the tree-based method make it possible to decode more than one bit per cycle [4], the tree-based approaches are not suitable for high performance applications such as high-definition television, because high clock rate processing is needed.

A *constant-output-rate* VLD decodes one symbol per cycle regardless of its length [5]. Generally speaking, a constant-output-rate VLD contains a look-up table which receives the variable-length code itself as the address. The decoded symbol (run-level pair or end-of-block) and the codeword length are generated in response to that address. Since the longest codeword excluding Escape has 17 bits, the LUT size could reach 131072 ($= 2^{17}$) words for a direct mapping of all possible codewords.

A *variable-input-output-rate* VLD is a mixture of the first two VLDs. It is implemented as a repeated table look-up, each step decoding a variable size chunk of bits. If a valid code was encountered, a run/level pair or an end-of-block is generated. If a miss is detected, a chunk size for the next look-up is generated. In this way, the short (most probable) symbols are preferentially decoded. A variable-input-output-rate VLD exhibits an acceptable decoding throughput, while the size of the look-up table is resonable small.

We conclude this section with a review on the architecture of the FPGA we used as an experimental reconfigurable core.

B. The FPGA architecture

Field-Programmable Gate Arrays (FPGA) [6] 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 configuration memory. In the sequel, we will assume that the architecture of the raw hardware is identical with that of an ACEX 1K device from Altera [7]. Our choice could allow future single-chip integration, since both ACEX 1K FPGAs and TriMedia are manufactured in the same TSMC technological process.

Briefly, an ACEX 1K device contains an array of Logic Cells, each including a 4-input Look-Up Table (LUT), a relative small number of Embedded Array Blocks, 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 EP1K10 device to 4992 logic cells and 12 EABs for EP1K100 device. The maximum operating frequency for synchronous designs mapped on an ACEX 1K FPGA is 180 MHz. More details regarding the architecture and operating modes of ACEX 1K devices, as well as data sheet parameters can be found in [7].

III. AN ARCHITECTURAL EXTENSION FOR TRIMEDIA/CPU64

TriMedia-CPU64 is a 64-bit 5 issue-slot VLIW core, launching a long instruction every clock cycle [8]. It has a uniform 64-bit wordsize through all functional units, the register file, load/store units, on-chip highway and external memory. Each of the five operations in a single instruction can in principle read two register arguments and write one register result every clock cycle. In addition, each operation can be guarded with an optional (4^{th}) register for conditional execution without branch penalty. The architecture supports subword parallelism and is optimized with respect to media-processing. With the exception of floating point divide and square root unit, all functional units have a recovery of 1, while their latency ranges from 1 to 4 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 [9], TriMedia-CPU64 processor is augmented with a Reconfigurable Functional Unit (RFU) which consists mainly of a reconfigurable array core. A hardwired Configuration Unit managing the reconfiguration of the raw hardware is associated to the RFU, as depicted in Figure 2. The reconfigurable functional unit 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, a kernel of new instructions is provided [9]. The new instructions are `SET` and `EXECUTE`. Generally speaking, the reconfiguration of the RFU is performed under the command of the `SET` instruction, while `EXECUTE` instructions launch the operations to be performed by the computing resources configured on the raw hardware [10].

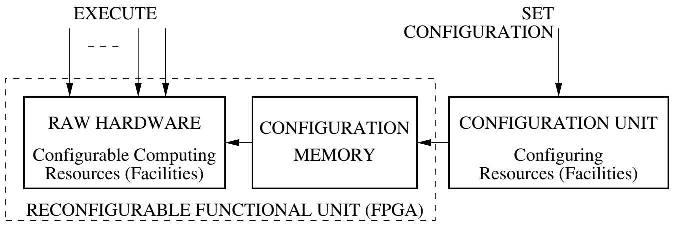

Fig. 2. Architectural extension for TriMedia-CPU64 VLIW core – [9].

The user is given a set of `EXECUTE` instructions encompassing different operation patterns: single- or multi-slot operations, operations with an immediate argument, etc. It is the responsibility of the user to choose the appropriate `EXECUTE` instruction corresponding to the pattern of the operation to be executed. Since the semantics of an operation performed by a RFU-mapped computing facility, its latency, recovery, and slot-assignment are all user-definable, the source code of the application should contain information to augment the Machine Description File [11]. This information is needed by the scheduler to schedule the newly defined operations, and can be specified by annotating the source code. For more details, we refer the reader to bibliography [9].

The next section will present the sintax and semantics of the VLD instruction, as well as implementation issues of the corresponding computing facility.

IV. VLD INSTRUCTION AND COMPUTING FACILITY

As with all hardwired computing resources, the latency of an RFU-configured computing resource should be known at compile time. Therefore, only a *constant-output-rate* VLD should be considered if the whole decoder was to be implemented on FPGA. With such decoder, no benefits from preferentially decoding the short (most probable) symbols can be achieved. In particular, we decided to configure on the FPGA a VLD computing resource which returns a DCT symbol (*run/level* pair or *end-of-block*) per call. A super-operation pattern with two input (Rx, Ry) and two output (Rz, Rw) registers is assigned to the variable-length decoder:

$$\textbf{VLD} \quad \textbf{Rx, Ry} \rightarrow \textbf{Rz, Rw}$$

The Rx register specifies the decoding parameters: AC/DC coefficient, luminance/chrominance block, intra/non-intra macroblock, MPEG-1/MPEG-2 compression standard, B14/B15 VLC decoding table [2]. The second register, Ry, contains 64 bits of the VL compressed data. The decoded symbol and its code length will be stored into registers Rz, while the register Rw will hold some control information.

TABLE I

VLD – THE FORMAT OF THE FIRST ARGUMENT (PARAMETER) REGISTER – RX (UINT32).

Field name	Acronym	Width (bit)	Position (bit)	Type (TriMedia)	Range	Description
Decoding parameters	dec_param	32	$31\ldots0$	uint32		
Not used	–	27	$31\ldots5$	n.a.	n.a.	
MPEG standard	mpeg_s	1	4	bit	$\{0,1\}$	= 1 for MPEG-2
Intra VLC format	i_vlc_f	1	3	bit	$\{0,1\}$	= 0 for B14 table
Intra/PB	intra_pb	1	2	bit	$\{0,1\}$	= 1 for intra macroblock
Luma/Chroma	y_c	1	1	bit	$\{0,1\}$	= 1 for luminance
DC/AC Coefficient	dc_ac	1	0	bit	$\{0,1\}$	= 1 for DC coefficient

TABLE II

VLD – THE FORMAT OF THE SECOND ARGUMENT REGISTER – RY (UINT64).

Field name	Acronym	Width (bit)	Position (bit)	Type (TriMedia)	Range	Description
MPEG string	–	64	$63\ldots0$	uint64	n.a.	The first bit of the MPEG string is the most-significant bit

TABLE III

VLD – THE FORMAT OF THE RETURNED VALUE IN REGISTER RZ (VEC64UB).

Field name	Acronym	Width (bit)	Position (bit)	Type (TriMedia)	Range	Description
Not used		32	$63\ldots32$	n.a.	n.a.	
Level	level	16	$31\ldots16$	int16		Extracted as two ui nt8.
Run	run	8	$15\ldots8$	uint8		
Code-length	code_length	8	$7\ldots0$	uint8	$0\ldots28$	

Since the VLD does not know the start of the next variable-length codeword until the current codeword is decoded, a new **VLD** operation can be launched only after the current one has completed. Consequently, a recovery lower than the latency gives no advantages, and such implementation should not be sought. The formats of the registers Rx, Ry, Rz, Rw are shown in Tables I, II, III, and IV.

Generally speaking, a constant-output-rate VLD computes the symbol code length by looking-up the 17 leading bits of the incoming bit stream into a look-up table. The decoder then sends the code length and the leading bits to other feed-forward circuitry for further decoding and immediately shifts the input by a number of bits equal with *code length*, to prepare the next decoding cycle. In cases where the number of symbols is large, there are some bits, referred to as *prefix*), that are common to long VLCs, called By exploiting these common prefixes, the size of the LUT can be reduced because the prefixes are no longer redundant in the LUT [12], [13], [14]. The basic idea of prefix precoding is to group the VLCs by their common prefixes, and to provide for LUTs, one for each group, which can decode codewords only in the corresponding group. The selection of the group is performed by means of additional circuits (an extra LUT in the mentioned papers).

Since a single EAB of an ACEX 1K device can implement a lookup table of 8 inputs, we partitioned the VLC table according to this FPGA architectural characteristic, as presented in Table V. In order to reduce the latency, the implementation of the VLD makes use of the *advanced computation*. The *run* and *level* for each and every group were decoded in parallel, as the valid symbol would belong to that group. Also in parallel, the code length of the symbol along with some *selection signals* are determined. Finally, the selection of the proper run and level pair is carried out. The implementation is presented in Figure 3.

Regarding the groups 1, 2, and 3, a number of 1, 6, and 9 leading bits are shifted out from the *same* VLC string, respectively. The three new resulted strings are each sent to a different EAB, and three run/level pairs are generated as if the shifted leading bits would have been those mentioned

TABLE IV

VLD – THE FORMAT OF THE RETURNED VALUE IN REGISTER RW (VEC64UB).

Field name		Acronym	Width (bit)	Position (bit)	Type (TriMedia)	Range	Description
Not used		–	32	$63\dots32$	n.a.	n.a.	
Not used		–	8	$31\dots24$	uint8	n.a.	
`Exit` controls		–	8	$23\dots16$	uint8		
	valid_decode	valid_decode	1	19	bit	$\{0,1\}$	$=1$ when valid decode
	error	error	1	18	bit	$\{0,1\}$	$=1$ when error
	EOB	EOB	1	17	bit	$\{0,1\}$	$=1$ when end-of-block
	exit_flag	exit_flag_1	1	16	bit	$\{0,1\}$	$=1$ when exit condition
Not used		–	8	$15\dots8$	uint8	n.a.	
`Exit` flag		–	8	$7\dots0$	uint8	$\{0,1\}$	
	exit_flag	exit	1	1	bit	$\{0,1\}$	$=1$ exit condition

TABLE V

THE PARTITIONING OF THE VLC CODES OF DCT COEFFICIENTS INTO GROUPS AND CLASSES.

Name of the group	No. of symbols in the class	Class / Leading bit-sequence	Code length	Bypassed bit-sequence	Effective address length
DC Group 0	2	1	1 + s	–	n.a.
End-of-block	1	10	2	–	n.a.
AC Group 0	2	11	2 + s	–	n.a.
Escape	1	0000 01	6 + 18/(14,22)	–	n.a.
Group 1	2	011	3 + s	0	3
	4	010	4 + s		4
	4	0011	5 + s		5
	2	0010 1	5 + s		5
	8	0001	6 + s		6
	8	0000 1	7 + s		7
	16	0010 0	8 + s		8
Group 2	16	0000 001	10 + s	0000 00	5
	32	0000 0001	12 + s		7
	32	0000 0000 1	13 + s		8
Group 3	32	0000 0000 01	14 + s	0000 0000 0	6
	32	0000 0000 001	15 + s		7
	32	0000 0000 0001	16 + s		8

in the column *Bypassed header*. By means of combinatorial circuits, the same procedure is carried out for groups 0, end-of-block, and escape.

Each of the leading bit-sequence which define the VLC class is decoded by a multiple-input gate. Once the class is detected, a multiplexer will select the proper output from the outputs of EABs, EOB detector, Escape detector, and Group 0 decoding. The code length of the decoded symbol is generated according to the detected class.

By simulation with Altera tools, we found that the FPGA-based VLD operation has a latency of 7 TriMedia cycles. 6 EABs and 16% of the logic cells of an EP1K100 device are used.

V. CONCLUSIONS

We have described the implementation of a VLD computing facility on a TriMedia/CPU64 reconfigurable functional unit. The VLD computing facility can decode one variable-length codeword per call, and exhibits a a latency of 7 TriMedia cycles. 6 embedded array blocks and 16% of the logic cells of an ACEX EP1K100 are used by the VLD computing facility.

REFERENCES

[1] Ming-Ting 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.

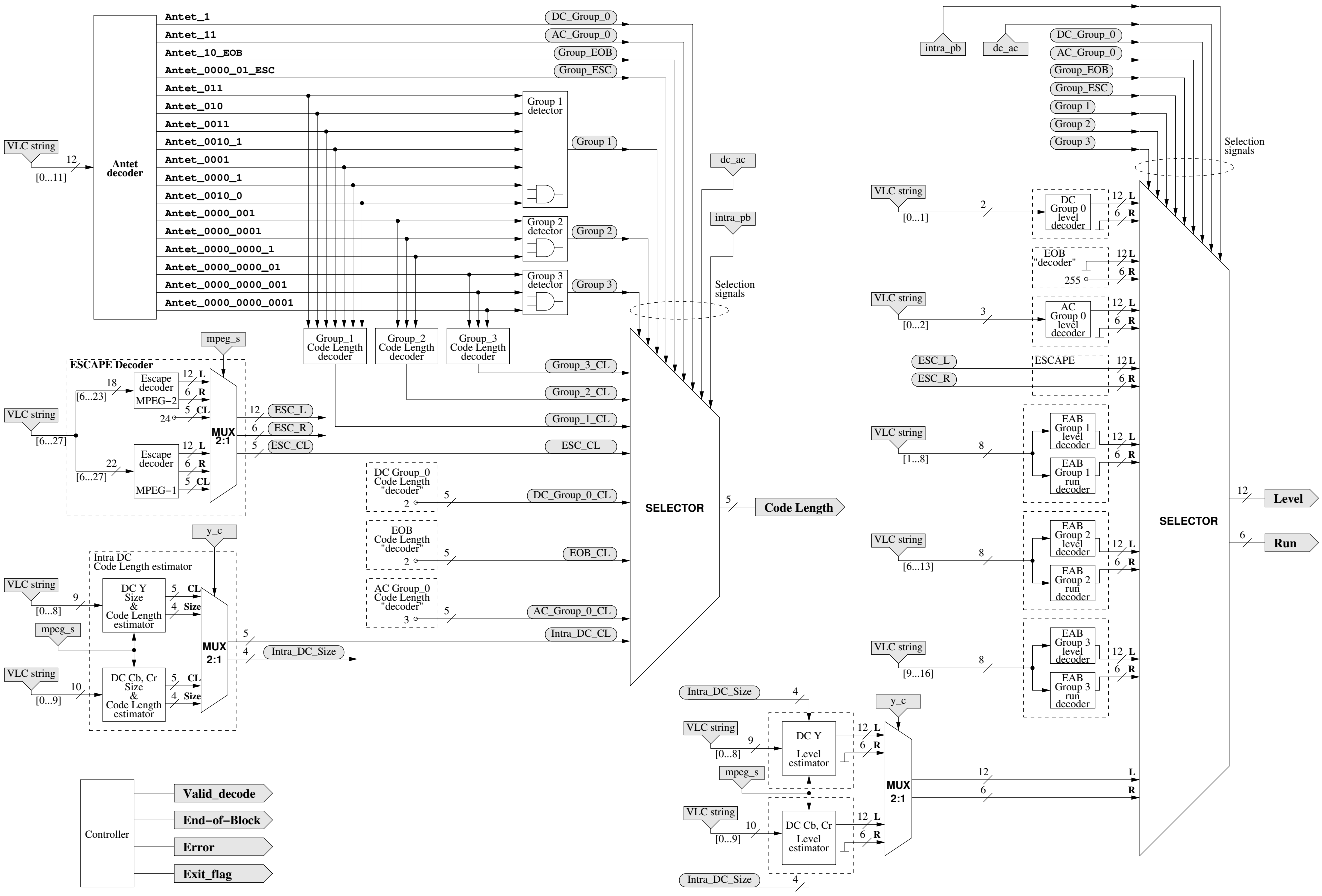

Fig. 3. **The VLD implementation on FPGA.**

[2] Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg, and Didier J. LeGall, *MPEG Video Compression Standard*, Chapman & Hall, New York, New York, 1996.

[3] Amar Mukherjee, N. Ranganathan, and M. Bassiouni, "Efficient VLSI Design for Data Transformation of Tree-Based Codes," *IEEE Transactions on Circuits and Systems*, vol. 38, no. 3, pp. 306–314, March 1991.

[4] Shigenori Kinouchi and Akira Sawada, "Variable Length Code Decoder," U.S. Patent No. 6,069,575, May 2000.

[5] Shaw-Min Lei and Ming-Ting 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.

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

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

[8] Jos T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E.-J. D. 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.

[9] Mihai Sima, Sorin Cotofana, Jos T.J. van Eijndhoven, Stamatis Vassiliadis, and Kees Vissers, "8×8 IDCT Implementation on an FPGA-augmented TriMedia," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Kenneth L. Pocek and Jeffrey M. Arnold, Eds., Rohnert Park, California, April 2001.

[10] Mihai Sima, Stamatis Vassiliadis, Sorin Cotofana, Jos T.J. van Eijndhoven, and Kees Vissers, "A Taxonomy of Custom Computing Machines," in *Proceedings of the First PROGRESS Workshop on Embedded Systems*, Utrecht, The Netherlands, October 2000, pp. 87–93.

[11] E.-J. D. Pol, B. J. M. Aarts, Jos T. J. van Eijndhoven, P. Struik, F. W. Sijstermans, M. J. A. Tromp, J. W. van de Waerdt, and P. van der Wolf, "TriMedia CPU64 Application Development Environment," in *Proceedings of International Conference on Computer Design*, Austin, Texas, October 1999, pp. 593–598.

[12] Seung Bae Choi and Moon Ho Lee, "High Speed Pattern Matching for a Fast Huffman Decoder," *IEEE Transactions on Consumer Electronics*, vol. 41, no. 1, pp. 97–103, February 1995.

[13] Kyeong-yuk Min and Jong-wha Chong, "A Memory-Efficient VLC decoder Architecture for MPEG-2 Application," in *IEEE Workshop on Signal Processing Systems. Design and Implementation*, Lafayette, Louisiana, October 2000, pp. 43–49.

[14] Seong Hwan Cho and Thucydides Xanthopoulos, "A Low Power Variable Length Decoder for MPEG-2 Based on Nonuniform Fine-Grain Table Partitioning," *IEEE Transactions on VLSI Systems*, vol. 7, no. 2, pp. 249–257, June 1999.