EFFICIENT DATA FLOW VARIABLE LENGTH DECODING IMPLEMENTATION FOR THE MPEG RECONFIGURABLE VIDEO CODING FRAMEWORK

Jianjun Li, Dandan Ding, Christophe Lucarz, Samuel Keller, Marco Mattavelli
Microelectronic Systems Laboratory (GR-LSM),
Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne 1015, Switzerland
Email: {david.li, dandan.ding, christophe.lucarz, samuel.keller, marco.mattavelli}@epfl.ch

ABSTRACT

In 2004, ISO/IEC SC29 better known as MPEG started a new standard initiative aiming at facilitating the deployment of multi-format video codec design and to enable the possibility of reconfiguring video codecs using a library of standard components. The new standard under development is called MPEG Reconfigurable Video Coding (RVC) framework. Whereas video coding tools are specified in the RVC library, when a new decoder is reconfigured choosing in principle any (sub)-set of tools, the corresponding bitstream syntax, described using MPEG-21 BSDL schema, and the associated parser need to be respectively derived and instantiated reconfiguration by reconfiguration. Therefore, the development of an efficient systematic procedure able to instantiate efficient bitstream parsing and particularly variable length decoding is an important component in RVC. This paper introduces an efficient data flow based implementation of the variable length decoding (VLD) process particularly adapted for the instantiation and synthesis of CAL parsers in the MPEG RVC framework.

Index Terms— Reconfigurable Video Coding Variable Length Decoding, CAL language, Huffman coding, Bitstream Syntax Description Language.

1. INTRODUCTION

Nowadays, video decoders need to support multiple codec standards because more and more video standards are deployed. Although different, all coding standards use the same or very similar coding tools and results to share similar architectures and implementations. Unfortunately, the way in which the existing coding standards are specified lacks of flexibility to adapt performances and complexity when new applications emerge. MPEG RVC standard intends to create a framework containing existing coding technology for developing, beside current standard decoders, new configurations for satisfying specific application constraints. RVC introduces a novelty since it promotes standardization at tool-level while maintaining interoperability between solutions from different implementers.

One challenge posed by the possibility of reconfiguring decoders is the need of appropriate procedures for the instantiation and synthesis of bitstream parsers in which efficient variable length decoding processes are important tasks. This paper presents a method for generating efficient components for the MPEG RVC library capable of decoding Variable Length codes. The components of the library like all other coding tools are CAL actors generated automatically given the input VLD table. By using the described procedure, VLD tables can be automatically and efficiently generated as FUs of RVC toolbox. By efficiently it is also meant that the data flow CAL FUs are suitable for efficient synthesis into SW and HW implementations.

The paper is organized as follows: the RVC framework is introduced in section 2. The variable length decoding toolbox is presented in section 3. Section 4 presents how to translate efficient VLD in CAL. Section 5 briefly introduces how to automatically generate a parser from a Bitstream Schema to CAL. Section 6 discusses about the hardware and software implementation of the parser and VLD tables. Section 7 concludes the paper.

2. MPEG RECONFIGURABLE VIDEO CODING OVERVIEW

MPEG has always worked to propose innovations in the video coding field that are capable of satisfying the changing landscape and needs of video coding applications. With this objective, MPEG intends to standardize the Reconfigurable Video Coding framework allowing a dynamic development, implementation and adoption of standardized video coding solutions based on a unified library of components with features of higher flexibility and reusability. RVC is a flexible framework for MPEG that tries to provide a systematic way of constructing video codecs from a collection of coding tools, it has been firstly presented in [3]. The goal of the introduction of such new interoperable model at coding tool level is twofold: to speed up the adoption and standardization of new technologies by adding new tools in toolbox and to enable the dynamic definition of new profiles. The modular data flow based specification formalism also provides a starting point for design that is adapted to yield direct synthesis of SW and HW by using appropriate tools, for direct mapping on SW and HW platforms.

A decoder specification under RVC is defined with the standard MPEG toolbox (instantiation and connections of the different coding tools) and the specification of the video bitstream syntax expressed in a MPEG-21 BSDL schema [9]. The toolbox consists of various coding tools which are also named Functional Units (FU). Each FU is a modular coding tool (such as IDCT, MC).

The concept of RVC framework can be illustrated by Fig. 1. The key difference between RVC and traditional codec standards is their conformance point. The traditional codec standards define their conformance point at decoder level whereas RVC defines it in tools level so that RVC enables much more flexibility and several configurations of components taken by previous monolithic specifications are possible.

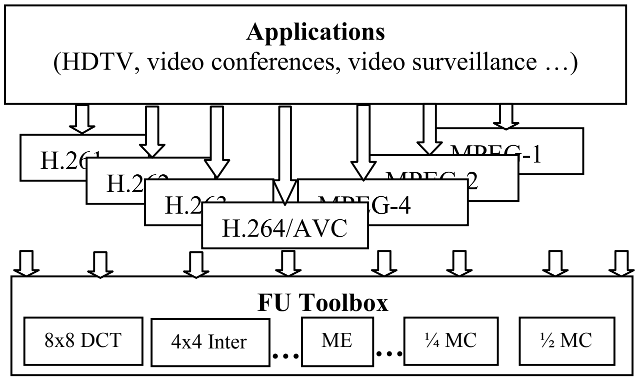

Fig.1. RVC framework

Another fundamental difference between RVC specification and the traditional standard codec specification is the data flow based formalism. In the traditional codec specifications, C/C++ is the language of the reference SW, which usually is composed by several thousands of lines and is getting more and more difficult to understand and to transform into efficient implementations. In the RVC framework, data flow actor-oriental language CAL [1] which is simpler, compact in terms of number of code lines, and does not include non necessary implementations details such as a fixed scheduling for C/C++ reference SW for instance, is used to describe FUs behavior.

3. VARIABLE LENGTH DECODING FOR THE RVC FRAMEWORK

One problem that needs to be solved when applying RVC is how to specify the parser that is in charge of decoding the bitstream of compressed video. In fact whereas all FUs of the standard MPEG toolbox are available under the form of CAL actors or as a proprietary implementation for specific platforms, the parser of a new decoder configuration need to be synthesized and instantiated automatically because it is a too burdensome task to let the designer write the parser actor in CAL. The parser is not considered as a coding tool because it does not contains any algorithm described by the standard. The unique task of the parser is to feed the coding tools with the right coded data contained in the bitstream. Therefore, a systematic procedure for synthesizing efficient parsers using appropriate FUs available in the standard toolbox is required.

3.1. Solutions for Variable Length Decoding

Variable length coding is the most popular entropy coding module which is used in many video and picture coding standards, such as JPEG, MPEG-x, and H.26x. One of the difficulties for RVC to describe variable length decoding is the large amount of tables. For example, in MPEG-4 SP [10] there are 8 tables and in MPEG-4 ASP [10], there are 19 tables. Including those tables directly in the syntax description (BSDL schema transmitted as header in the bitstream) would imply inefficiency in the compactness of the description of a new codec configuration, but would also requires large memory and bandwidth. Another difficulty is the parsing process of the undefined bit-length of syntax. In order to avoid carrying VLD tables in bitstream description, VLD tables could be separated and implemented in CAL as FUs of RVC toolbox. The proposed Huffman decoding method is applied to VLD tables which further improved efficiency. The bitstream syntax parser is generated automatically as an independent FU in CAL language from a XML schema describing the structure of the bitstream. The transformation process is implemented using XSLT. The bitstream schema is specified in a XML dialect called Bitstream Syntax Description Language (BSDL) [9], a MPEG-21 standard. Negotiation between the syntax parser and VLD tables are also established in XSLT for variable decoding process. The systematic solution for syntax parser is highly efficient and flexible to decode a reconfigured Bitstream.

3.2. Efficient Huffman Decoding method

In this section, a CAL model for efficient Huffman decoding is proposed for VLD tables of MPEG-2 and MPEG-4. The proposed implementation is optimized aiming at searching time and memory requirement reduction.

Huffman coding has been adopted by MPEG-2 and MPEG-4 entropy coding. Sets of codewords are defined based on the probability distributions of "generic" video material. The direct way to decode variable length syntax is using a full search method:

1) The variable length decoder receives one bit from Bitstream.
2) Look through the corresponding table from the beginning to check whether it is coincide with certain code.
3) If it is found, output the value from the table.
4) Or else receive another bit and combine it with the former bits, go back to step 1).

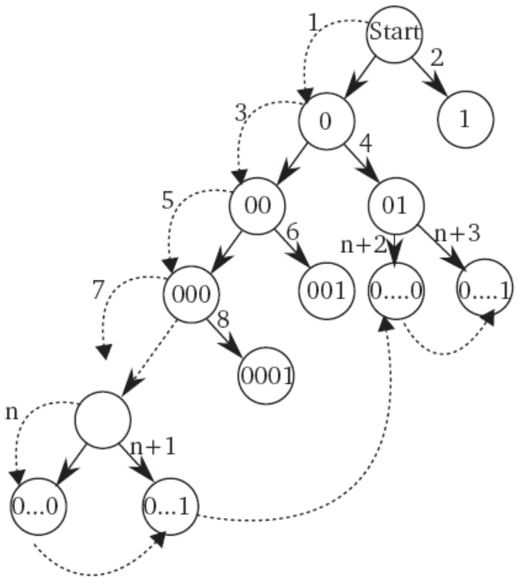

Fig.2. VLD binary searching tree

Such full search method is simple but not efficient enough because of duplicate lookup every time one bit is received. In addition, it requires a 2-D memory for each table which is not a good choice for hardware implementation.

The proposed method rearranges the code in the Huffman tree. The binary Huffman tree searching can find the optimal route in short time and requires less medium data. As shown in Fig. 2, the variable length coding codeword starts with the first incoming bit. The current bit goes to the left leaf if the coming bit is "0". Otherwise, it goes to the right leaf while "1". Weight of each leaf is marked with the same value of lookup index for corresponding VLD tables. That is to say, every time one bit is consumed at input, one index is generated and one lookup result is generated as output. If the result is a true decoded value, it is provided to the output of the CAL FUs and the search of the variable length coding is completed. On the other hand, if the result is a false decoded value, a further searching is continued until a completed codeword is found.

Different video coding standards have different VLD tables. Even in a single standard, different profiles and levels have different VLD tables' scope. The most efficient solution for the RVC framework would be to build separate FUs available in the standard toolbox for each VLD table decoding and then generate dynamically a parser as composition of a synthesized CAL parser and VLD decoding FUs.. Each VLD table is considered as an independent FU of the RVC toolbox. For example, in MPEG-4 specification Annex B, there are 8 VLD tables that are used by a simple profile decoder. They are B-6, B-7, B-8, B-12, B-13, B14, B-16 and B-17. In the MPEG-4 advanced simple profile, to these tables other VLD tables are needed. It is unnecessary to generate them again, just access to toolbox and get related FUs. Take MPEG-4 SP for example, we generate the VLD FUs and name them with the table name, such as B-6, B-7 and so on, as showed in Fig.5.

Code	**mbtype**	**cbpc(56)**
1	3	00
001	3	01
010	3	10
011	3	11
0001	4	00
000001	4	01
000010	4	10
000011	4	11
0000 0000 1	Stuffing	--

Table-1. Example of VLC table B-6 for *mcbpc*

Input File 1	Input File 2	Output File
1 001 010 011 0001 000001 000010 000011	3 19 35 51 4 20 36 52	Table name: B6 Start index: 0 10, 12, 18, 58, 26, 76, 34, 16, 42, 50, 1, 80, 144, 208, 140, 204

Table-2 Generated VLD table for B-6

MPEG-4 specification Annex B Table B.6 [10] is listed as above, which is the VLD table for *mcbpc* for I-VOPs and S-VOPs. In this table, the eight values refer to different chroma coded block pattern (*cbp*) of block 4 and 5. Table-2 is the generated VLD table by the proposed method. All the data with underscore in Table-2 are media data, which means that this is not true decode value and the VLD table engine will keep search the next value when the underscore data is found. The VLD table engine will stop and report searching failure if “1” is found, which means an error code is detected. Otherwise, true decoded value from VLD table is returned and the decoding process for the syntax is completed.

4. MODELING VARIABLE LENGTH DECODING OF MPEG-4 SP IN CAL

CAL [1] is a dataflow and actor-oriented language, specified as a part of the Ptolemy II project at the UC Berkeley. CAL language has concise syntax structure and is suitable for specifying complex signal processing systems as MPEG decoders.

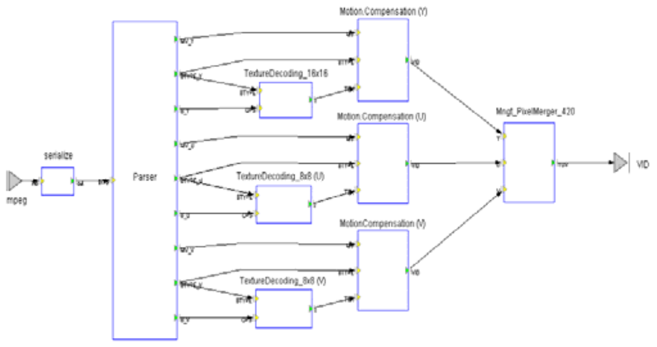

Fig.3. High level view of the CAL model of the MPEG-4 SP decoder

Fig.3 shows the graphical representation of the CAL model of the MPEG-4 SP decoder [10]. The Open Dataflow environment [6] is used to design and simulate CAL models. The decoder includes several networks of actors. The incoming bitstream is at first converted into sequential bits by the “*serialize*” FU, and then is decoded by the “*Parser*”. The “*TextureDecoding*” and “*MotionCompensation*” networks of actors contain all the coding tools necessary for decoding the video.

Figure 4 illustrates the inside of the “*parser*” FU present in figure 3. It shows how VLD FUs are connected to the parser for decoding Variable Length codes. For the sake of clarity, figure 4 represents only the connection of one VLD FU to the parser. This VLD FU serves at decoding the DCT coefficients (table B-16 in Annex B of the MPEG-4 standard [10]). The FU “*parser*” is generated automatically by the XSLT process (see section 5). The VLD FU is generated using the process described in section 3. The “*BlockExpand*” FU is part of the MPEG toolbox. It outputs the AC coefficients.

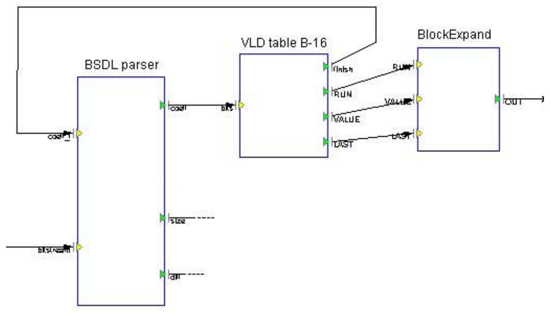

Fig.4. Connections of the parser to a VLD Functional Unit

When the parser meets a Variable Length code, it consumes only one bit from the bitstream port. It sends it to the VLD FU. If there is no entry in the table which corresponds to the input bit, the VLD FU sends back to the parser a token noticing that no matching has been found. Thus, the parser consumes an additional bit and sends it to the VLD FU. This latter will check if the first bit and the newly received bit match an entry in the table. If no, it continues sending token to the parser, saying that there is no matching and the parser must send an additional bit. If yes, the VLD FU sends a token to the parser saying that a matching has been found and the parser can parse the next element of the bitstream. The result of the parsing is then outputted by the VLD FU to the “*BlockExpand*” FU.

The source code of the VLD FU for decoding the “*mbcpc*” variable code is shown in Fig.5. The only part of the FU which is automatically generated is a list of numbers, representing the VLD table. The rest of the code is always the same for all the VLD FUs. The extra code is needed to handle the optimized list of number representing the VLD table.

```
import all caltrop.lib.BitOps;

actor VLD_mcbpc_intra(int VLD_DATA_SZ, int VLD_ADDR_SZ)
  string bits ==> int(size=2) finish, int(size=VLD_DATA_SIZE) data:

  int START_INDEX = 0;
  int( size=VLD_ADDR_SZ ) vld_index;
  int( size=VLD_DATA_SZ ) vld_codeword := 1;

// ********** automatically generated part ********
list( type:int( size=VLD_DATA_SZ ), size=16 )
vld_table = [ 10, 12, 18, 58, 26, 76, 34, 16, 42, 50, 1, 80, 144,
208, 140, 204 ];
// ************************************************

  procedure start_vld_engine( int index )
  begin
    vld_index := index;
    vld_codeword := 2;
  end

  function vld_success() --> bool: bitand(vld_codeword,3) = 0 end
  function vld_continue() --> bool: bitand(vld_codeword,3) = 2 end
  function vld_failure() --> bool: bitand(vld_codeword,1) = 1 end
  function vld_result() --> int( size=VLD_DATA_SZ ):
rshift(vld_codeword,2) end

  start_VLD: action ==>
  do
    start_vld_engine( START_INDEX );
  end
  read_in_bits: action bits:[b] ==>
  do
   vld_codeword := vld_table[ vld_index + if b="1" then 1 else 0
end ];
    vld_index := rshift(vld_codeword,2);
  end

  continue_VLD: action ==> finish:[f]
  guard
    vld_continue()
  var
    int(size=2) f := 0
  end

  fail_VLD: action ==>
  guard
    vld_failure()
  do
    println("VLD FAILURE");
  end

  finish_VLD: action ==> finish:[f], data:[d]
  guard
    vld_success()
  var
    int(size=2) f := 2,
    int(size=VLD_DATA_SZ) d := vld_result()
  end

  schedule fsm start_VLD:
    start_VLD        ( start_VLD    ) --> read_in_bits;
    read_in_bits     ( read_in_bits ) --> process;
    process          ( continue_VLD ) --> read_in_bits;
    process          ( fail_VLD     ) --> start_VLD;
    process          ( finish_VLD   ) --> start_VLD;
  endschedule

endactor
```

Fig.5. CAL source code of a VLD Functional Unit

This section showed how the Variable Length Decoding process has been modeled in CAL. The next section shows how the parser handles the communications with the VLD FUs to decode these variable length codes.

5. FROM BITSTREAM SCHEMA TO PARSER

Video coding is used under the various multimedia applications such as video conferencing, digital storage media, television broadcasting, and internet streaming. Due to the heterogeneity of modern networks and terminals, current multimedia technology has to deal with different user's requirements. As such, the use of scalable video coding, which derives useful video from subsets of a bitstream, is a must. RVC is compatible with SVC very well and it can implement SVC in function unit level. At this moment, the solution is that the MPEG-21 multimedia framework enables transparent and augmented use of multimedia resources across a wide range of networks and devices used by different communities [4].

The BSDL parser is a primordial Functional Unit in the RVC framework because it feeds the coding tool chain with the information contained in the bitstream to be decoded. As RVC is a framework for rapid development of decoding solution, the structure of the bitstream can be modified in order to explore the design space. To avoid the designer to write it by hand (which would be very time-consuming and error prone), a method has been developed to generate directly a parser from the bitstream syntax [3]. Figure 6 shows the components of this transformation process. Each component is implemented in a separate XSLT stylesheet.

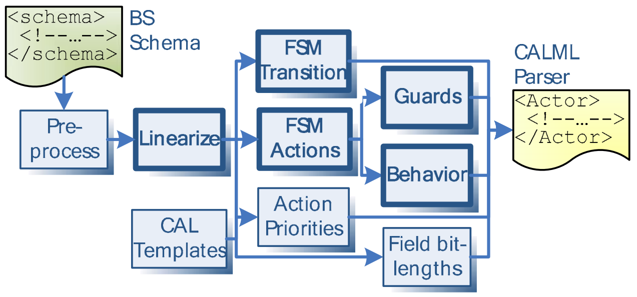

Fig.6. XSLT transformation process: from BSDL to CAL

Pre-processing is the first operation conducted by the top level stylesheet. The pre-processing collects the individual schemata into a single intermediate tree, taking care to correctly manage the namespace of each component Schema and also performs a number of other tasks, including assigning names to anonymous types and structures. Finite State Machine (FSM) design is the major component of the parser actor. The FSM schedules the reading of bits from the input Bitstream into the fields in the various output structures, along with all other components of the actor. The FSM is specified as a set of transitions, where each transition has an initial state, a final state, and an action. BSDL specifies that the order of options within a choice establishes their priority: the first option has priority over the second, and so on. These priorities are recorded in the actor as priorities between the test actions. Guard expressions are built from the control-flow constructs in the BSDL Schema. The Behaviour of each action is to complete such tasks as storing data in the appropriate location in the output structure. Finally, the CAL component declares templates for each of the constructs in the language, such as an FSM schedule, a function call, or an assignment. These templates are called by other components of the stylesheet when building the actor. Collecting all of the CAL syntax into a single stylesheet also means that an alternative stylesheet could be provided in place of the CAL sheet.

Figure 7 illustrates a part of the parser automatically generated from the bitstream schema. It shows the actions and the finite state machine generated for handling the communication between itself and external VLD FUs. When the parser meets a variable length code, the actions shown in figure 8 are generated. First, the parser reads one bit from the bitstream input port (`DCT_Coeff.read` action). The next step consists in sending the bit to the corresponding VLD table; it is done in action `DCT_Coeff.output`. Then, the parser waits for a token coming from the VLD FU. This token (`finish`) indicates if a matching has been found in the table or not. If yes, the value of `finish` is true and the action `DCT_Coeff.finish` is fired and the number of bits to read for the next element is set. If not, the value of `finish` is false and the `DCT_Coeff.notFinished` is fired and one more bit must be read (`M4V_VLC_LENGTH = 1`). The finite state machine summarizes the transitions.

```
DCT_Coeff.read: action  ==>
guard
  readDone()
end

DCT_Coeff.output: action  ==> B16: [current]
do
  current := read_result_in_progress ;
end

DCT_Coeff.finish: action B16_f: [finish] ==>
guard
 finish
do
 setRead(M4V_NEXT_ELEMENT_LENGTH);
end

DCT_Coeff.notFinished: action B16_f: [finish] ==>
guard
 not finish
do
 setRead(M4V_VLC_LENGTH);
end

[…]

// Finite State Machine
Previous_state      (previous_action)        --> DCT_Coeff_exists;
DCT_Coeff_exists    (DCT_Coeff.read)         --> DCT_Coeff_output;
DCT_Coeff_output    (DCT_Coeff.output)       --> DCT_Coeff_result;
DCT_Coeff_result    (DCT_Coeff.notFinished)  --> DCT_Coeff_exists;
DCT_Coeff_result    (DCT_Coeff.finish)       --> Next_state;
```

Fig.7. Source code of the automatically generated parser for the negotiation between the parser and the VLD FU

This section showed how the variable length decoding process is handled by the generated parser to decode variable length codes.

6. HW AND SW IMPLEMENTATION

The important reason for which CAL has been adopted as language specifying the reference software of the RVC toolbox is that CAL is suitable for direct synthesis of "efficient" software and hardware by means of CAL2SW and CAL2HW tools [7,8]. Furthermore, the very interesting aspect of this framework is that CAL models are used as inputs both for the hardware and software code generators. Thus software and hardware implementations can be derived from a unique CAL model. The designer develops an unique model and can generate seamlessly hardware and software implementation of CAL actors.

As the code of the VLD actors and parser are very simple, the generation of efficient code is straightforward. In [8], it has been shown that the hardware implementation of the MPEG-4 SP decoder modeled in CAL is more efficient than the one designed by hand in VHDL. Furthermore, in terms of coding effort, it took twice less time for a designer to write the CAL model than the VHDL model.

7. CONLUSION

Reconfigurable video coding framework is introduced in this paper. An efficient VLD toolbox can be generated by the proposed design. It is successfully implemented in CAL and validated by simulations. This paper shows that it is possible to dynamically generate a RVC parser using a BSDL description of the Bitstream and assembling RVC decoding FUs from the standard RVC toolbox.

8. REFERENCES

[1] J. Eker and J.W. Janneck, "CAL Language Report," Tech. Memo UCB/ERL M03/48, UC Berkeley, 2003.

[2] Jer-Min Hsiao ,Chun-Jen Tsai "Analysis of an SOC architecture for MPEG reconfigurable video coding", ISCAS 2007, May 2007.

[3] C. Lucarz et al., "Reconfigurable media coding: a new specification model for multimedia coders," Proceedings of the IEEE Workshop on Signal Processing Systems, 2007. Pages: 481 - 486.

[4] W. De Neve, F. De Keukelaere, K. De Wolf, R. Van de Walle, "Applying MPEG-21 BSDL to the JVT H.264/AVC specification in MPEG-21 Session Mobility scenarios," 5th International Workshop on Image Analysis for Multimedia, April 2004.

[5] J. Thomas-Kerr*, et al.*, "An efficient approach to generic multimedia adaptation," presented at *Multimedia, 14th ACM Intl. conf. on*, 2006.

[6] The Open DataFlow environment on Sourceforge *http://opendf.sourceforge.net/*

[7] Matthieu Wipliez, Ghislain Roquier, Mickaël Raulet, Jean-Francois Nezan, and Olivier Déforges, "Code generation for the MPEG reconfigurable video coding framework: from CAL actions to C functions," in IEEE International Conference on Multimedia & Expo (ICME), Hannover, Germany, 2008 (to appear).

[8] Jörn W. Janneck, Ian D. Miller, Dave B. Parlour, Marco Mattavelli, Christophe Lucarz, Matthieu Wipliez, Mickaël Raulet, and Ghislain Roquier, "Translating Dataflow Programs to Efficient Hardware: an MPEG-4 Simple Profile Decoder Case Study," in Design, Automation and Test in Europe (DATE), Munich, Germany, 2008.

[9] ISO/IEC 23001-5, Bitstream Syntax Description Language.

[10] ISO/IEC14496, Coding of audio-visual objects (MPEG-4).