CAL Dataflow Components For an MPEG RVC AVC Baseline Encoder

Hussein Aman-Allah, Karim Maarouf, Ehab Hanna, Ihab Amer, Marco Mattavelli

Laboratory of Microelectronic Systems (GR-LSM), EPFL
CH-1015 Lausanne, Switzerland

Hussein Aman-Allah:
Email: hussein.aman-allah@epfl.ch
Tel: +41 21 69 30928
Fax: +41 21 69 34663

Karim Maarouf:
Email: karim.maarouf@epfl.ch
Tel: +41 21 69 30928
Fax: +41 21 69 34663

Ehab Hanna:
Email: ehab.hanna@epfl.ch
Tel: +41 21 69 30928
Fax: +41 21 69 34663

Ihab Amer:
Email: ihab.amer@epfl.ch
Tel: +41 21 69 30929
Fax: +41 21 69 34663

Marco Mattavelli:
Email: marco.mattavelli@epfl.ch
Tel: +41 21 69 36984
Fax: +41 21 69 34663

Abstract. *In this paper, an efficient H.264/AVC baseline encoder, described in RVC-CAL actor language, is introduced. The main aim of the paper is two folds: a) to demonstrate the flexibility and ease that is provided by RVC-CAL, which allows for efficient implementation of the presented encoder, and b) to shed light on the advantages that can be brought into the RVC framework by including such encoding tools. The main modules of the designed encoder include: Inter Frame Prediction (Motion Estimation/Compensation), Intra Frame Prediction, and Entropy Coding. Descriptions of the designed modules, accompanied with RVC-CAL design issues are provided. A comparison between different development approaches is also provided. The obtained results show that specifying complex video codecs (e.g. H.264/AVC encoder) using RVC-CAL followed by automatic translation into HDL, which is achievable by the tools that support the standard, results in more efficient HW implementation compared to the traditional HW design flow. A discussion that explains the reasons behind such results concludes the paper.*

Keywords: Reconfigurable Video Coding, MPEG Video Tool Library, CAL Actor Language, H.264/AVC baseline profile.

1. Introduction

1.1 RVC Standard

The MPEG RVC standard main aim is "to offer a more flexible use and faster path to innovation of MPEG standards in a way that is competitive in the current dynamic environment" [1]. This is meant to give MPEG standards an edge over its market competitors by substantially reducing the Time to Market (TTM). The RVC initiative exploits the reuse of obvious commonalties among different MPEG standards and their possible extensions using appropriate higher level formalisms. Thus the objective of the RVC standard is to describe current and future codecs in a way that makes such commonalities explicit, reducing the implementation burden for device vendors [2]. In order to achieve this objective, RVC suggests simplifying the specification of new coding tools by reusing components of previous standards instead of defining new ones.

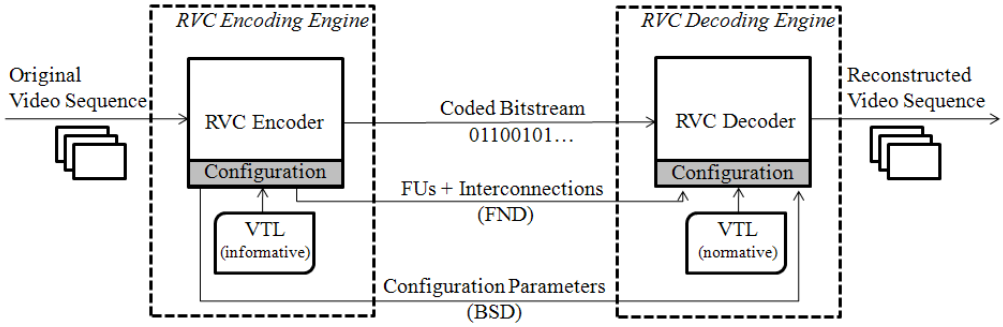

Figure 1 - MPEG RVC Encoding/Decoding Scenario

Figure 1 [3] defines the essential elements of the RVC framework. It consists of three major components: a normative Video Tool Library (VTL), a normative description of the connections between the coding tools (i.e. Functional Units or FUs), namely Functional unit Description (FND), which is written in the Functional unit Network Language (RVC-FNL), and a normative description of the structure of the bitstream, namely Bitstream Syntax Description (BSD), which is written in the Bitstream Syntax Description Language (RVC-BSDL) [1].

The VTL is specified with a textual normative specification and a corresponding reference SW. The SW specification is implemented in RVC-CAL, which is standardized by ISO/IEC MPEG and differs from the original CAL specification by limiting a set of data types and operators. This enables the existence of more efficient CAL2SW and CAL2HW tools.

CAL is a dataflow oriented language, specified in 2003 at the University of California at Berkley. It is a textual programming language that defines the functionality of dataflow components referred to as *actors.* Each actor is a modular component with a state. Only the actor itself is allowed to modify its state and the interaction between different actors takes place through FIFO channels connecting the *output ports* of one actor to the *input ports* of another. Data is exchanged between actors in the form of *tokens* [4]. Hence, CAL exhibits strong abstraction and

encapsulation properties that allow for the definition of the FUs so that they can be interchangeably combined and connected to form different video codecs. CAL also facilitates concurrent development because the maintainability and understandability of the code are improved over other sequential models implemented, for example, in C/C++.

Actors within a CAL program execute in sequences of *transitions.* During each of these, an actor may *consume* an input token, *modify* its internal state or *produce* an output token. Every actor is described in a set of *actions*. Each action defines a set of transitions to be performed under certain conditions depending on the availability of input tokens, their values, the internal state of the actor and the priority corresponding to that action. CAL supports several language constructs that describe actions firing process, including *action guards*, *finite state machines* and *action priorities* [4].

RVC-CAL, which is a subset of CAL, avoids the concerns that arise with other languages due to the ever-changing coding styles. It only provides natural constructs that have been identified by the RVC framework as essential elements for building MPEG codecs [4]. It also abstracts the majority of the algorithm-irrelevant details (like the clock synchronization, handshaking protocols, etc) allowing the developers to better focus on the implementation and refinement of the algorithm in consideration, which leads to results of much higher degrees of productivity and efficiency as objectively quantified in section 5.

1.2 H.264/AVC Baseline Profile

H.264/AVC standard specifies different profiles, baseline, main, extended and a range of high profiles, in addition to the scalable profiles. Each profile specifies a particular set of components. The baseline targets the broader band of applications, including video conferencing, wireless video, etc [5]**.** It supports inter and intra frame coding and entropy coding with Context Adaptive Variable Length Coding (CAVLC). Figure 2 provides an overview of a baseline H.264/AVC encoder showing the different modules: inter frame prediction (Motion Estimation and Motion Compensation), intra frame prediction, a deblocking filter, forward and inverse transforms (T and T^{-1}), forward and inverse quantization (Q and Q^{-1}) and entropy coding.

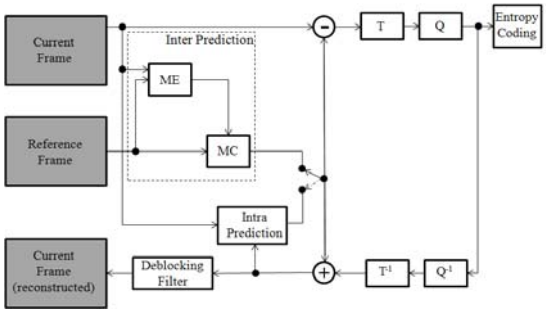

Figure 2 - H.264/AVC Baseline Profile Encoder Block Diagram

1.3 RVC Encoding Tools

In this paper, an RVC-CAL implementation (targeting hardware) of an H.264/AVC baseline profile encoder is presented. The purpose is to verify the suitability of RVC-CAL (together with the RVC supporting technologies) to describe (and implement) such complex video coding systems (e.g. H.264/AVC encoder). In addition, the presented design modules can be used as building blocks to demonstrate that the existence of RVC encoding tools supports the evolution of the RVC standard. Many benefits can be achieved by supporting the RVC framework with such encoding tools. This includes (but is not limited to):

- Instead of modifying the available C/C++ software reference model of a specific MPEG standard to make it able to generate the BSDL bitstream and the FNL decoder description in order to test the conformance of a corresponding RVC decoder, building an RVC encoder using RVC-CAL is more convenient. Especially due to the dynamic nature of the emerging RVC standard that requires high degree of flexibility in bitstream generation.
- Building the encoder using RVC-CAL enables the usage of the commonalities between many components within various MPEG standards. Hence, the existence of an informative VTL for encoders may be advisable.
- This allows for the construction of “configurable encoders” using the encoder’s informative VTL, which specifies the set of functional units (FUs) that may be

interchangeably combined and connected to form different video encoders, with various compression performances and implementation complexities. Such feature is highly demanded in many state-of-the-art broadcasting facilities, such as those that rely on simulcast and/or Scalable Video Coding (SVC).

- Some of the FUs of the normative VTL of the RVC decoder can be used to construct the encoder, either directly (such as the idct module), or indirectly, by inferring the IO structure of a module in the encoder from the corresponding module in the decoder.

Many components of the designed encoder represent potential contributors to the RVC VTL. The paper previews the design of the major components comprising the encoder and compares the development effort and productivity against other software and hardware approaches.

The remainder of this paper is organized as follows: Sections 2, 3 and 4 discuss inter, intra frame prediction, and entropy coding respectively with emphasis on the RVC-CAL design and implementation. Analysis of the proposed approach and comparison of results against other development approaches is provided in section 5**.** Finally, section 6 concludes the paper.

2. Inter Prediction

2.1 Motion Estimation

H.264/AVC uses block based motion estimation (ME) combined with transform coding for compressing video. By using block based ME the motion occurring between the frames can be estimated, thus eliminating the temporal redundancy between frames which allows for high compression [6].

A dataflow implementation of the ME/MC using RVC-CAL actor language is introduced. Full search is chosen to be the search algorithm in this case study as it exploits the parallel nature of hardware and the metric of comparison is the sum of absolute differences (SAD). The abstract RVC-CAL implementation presented allows for re-configurability and interchangeability by adding other search algorithms and metrics as well as other tools of the H.264/AVC such as sub-pixel ME. Figure 3[3] describes the ME RVC-CAL network consisting of 14 interacting actors.

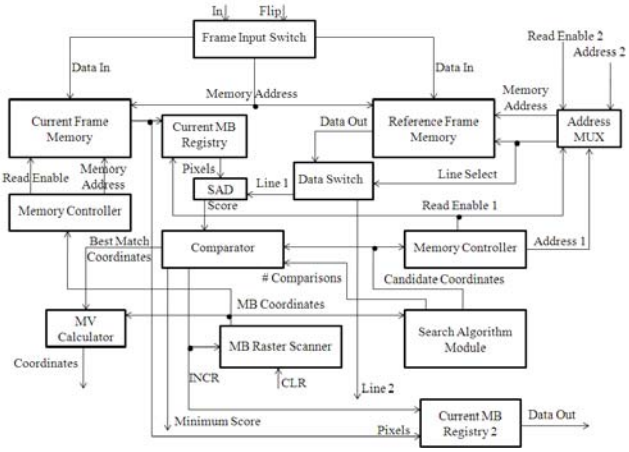

Figure 3 – Motion Estimation RVC-CAL Network

The *Frame Input Switch* actor reads the frames of a raw video sequence byte-wise and makes them available in the form of tokens. Each token is consumed and stored in a frame memory corresponding to the frame to which the token belongs. The *Current Frame Memory* and the *Reference Frame Memory* represent the two frame memories. Once both memories have been fully loaded the module begins the ME process. Two *Memory Controllers* control both memories by passing the proper addresses to extract the pixels of a macroblock (MB) from a frame memory.

The *Macroblock Raster Scanner* reads all MBs in the current frame in a raster fashion, and then produces two tokens representing the starting x and y positions of the first MB. The *Current Frame Memory controller* consumes the x and y tokens and produces address tokens corresponding to the pixels of that MB from the current frame memory.

The S*earch Algorithm Module* consumes the same x and y tokens containing the location of the original MB. The actor consumes the tokens, calculates a search window around that location, chooses candidate MBs and passes their locations to the *Reference Frame Memory Controller*. In this case all candidate MBs in the search window are chosen since the full search algorithm is in use.

When both *Memory Controllers* consume the tokens, the Current Frame Controller receives the location of the original MB, and the reference frame controller receives the locations of the candidate blocks. The implementation does not define any form of synchronization as it is handled autonomously during synthesis to hardware description language (HDL). Both controllers instruct the memories to output the pixels of the MBs. The current frame memory passes the pixels of the original block to the *Current MB Register*. Since the original block is used in all comparisons; it is stored in a register in order to reduce memory accesses. The *Reference Frame Memory* produces pixel tokens of the candidate blocks directly to the *SAD* actor, which performs the SAD operation. The *SAD* consumes the pixel tokens from the reference frame memory and the *Current MB Register* and calculates the score. The score is then passed to the *Comparator* which chooses the best score.

Once all candidate MBs have been compared with the original block the *Comparator* outputs the location of the best scoring block to the *Motion Vector Calculator*; which calculates the motion vector for the original MB by subtracting the best score's location from the original block's location. The best score and the motion vector tokens are then output to the *Motion Compensation Module* which uses the motion vectors along with pixels from the *Reference Frame Memory* in constructing the *compensated frame* and finally to calculate the *compensation error*. The previous process is repeated until all blocks in the current frame have been matched.

2.2 Motion Compensation

Motion compensation (MC) is the process where the current frame is reconstructed from the previous frame. This is done by using pixels of the reference frame from locations specified by the motion vectors obtained in the ME process. The result is a motion compensated frame which is a prediction of the motion that occurred between the reference and the current frame.

In Figure 3, the three actors: *Address Mux, Data Switch* and *Current MB Register 2* are used by the MC module. The *Address Mux* takes as input external addresses supplied by the MC module and internal addresses supplied by the *Reference Frame Memory Controller* and their enable signals. The external addresses represent the pixels of a MB that are to be used in constructing the compensated frame. The function of the *Address Mux* is to pass the addresses to the memory and to control the *Data Switch* actor via the Line select signal. The *Data Switch* outputs Data from the *Reference Frame Memory* to either the *SAD* actor or to the MC module based on its line select input. The *Current MB Register 2* only fires when a best match is selected, the current MB is needed in calculating the compensation error at the MC module.

Figure 4 [3] shows the structure of the MC module. The module contains three actors that are in the ME module along with four new ones. An external controller initializes the module by producing a CLR token to the *Macroblock Raster Scanner* which, in turn, produces two tokens, x and y. *Memory Controller 1* retrieves the MBs of the compensated frame by consuming the x and y tokens after the *Adder* compensates them with the motion vector. Concurrently, *Memory Controller 2* specifies where to write in the memory. *Memory Controller1* produces the address tokens that are consumed at the ME module by the *Address Mux* which as a result extracts and returns the compensated MB pixels to the MC module.

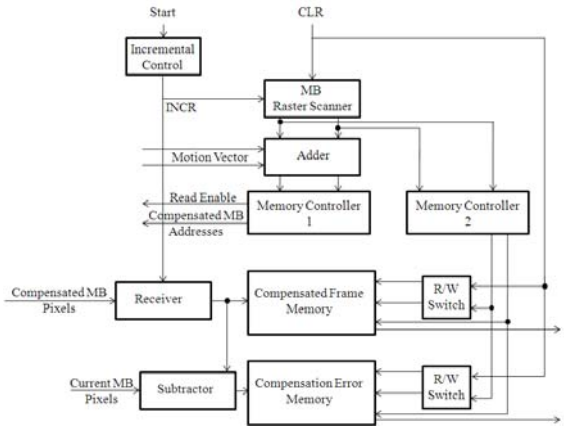

Figure 4 - Motion Compensation RVC-CAL Network

The *Receiver* actor consumes the compensated MB pixels and produces increment (INCR) tokens to the *Macroblock Raster Scanner* to fetch the next MB. It then relays the pixels to be stored in the *Compensated Frame Memory*, and to the *Subtractor*. The *Subtractor* stores the difference between compensated MB pixels and the current MB pixels in the *Compensation Error Memory*. After both Memories are fully loaded, another CLR token is sent to the *Macroblock Raster Scanner* to start reading data from the memories. The CLR token also flips the *R/W Switch* actors from write to read.

Figure 5 shows the interaction between the ME and MC networks. After the *compensated frame* is constructed and the *compensation error* is calculated, the compensation error pixels are sent to the transform coding module for discrete cosine transform (DCT), quantization and rescaling. The current frame is reconstructed by adding both compensated frame and error after quantization and rescaling. It is then passed back to the ME module as the reference frame and a new frame is read from the video sequence as the new current frame.

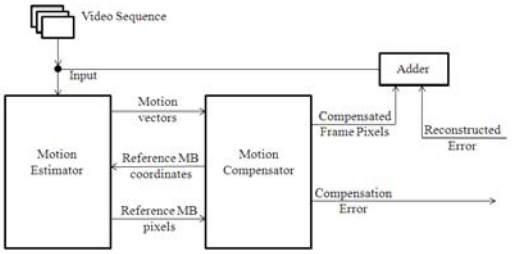

Figure 5 - ME and MC Networks Interaction

3. Intra Prediction

During intra prediction, a predictor P is formed from previously encoded MBs. In H.264/AVC luma prediction, P can be formed either for every 4x4 block in a MB (16 in total) or for the 16x16 MB as a whole. There are nine prediction modes for 4x4 and four for 16x16 block sizes. The encoder selects the mode which minimizes the difference between P and the original block to be encoded [5]. Figure 6 shows a 4x4 luma block in which 'A-M' represent the previously encoded samples and 'a-p' represent the values to be predicted.

M	A	B	C	D	E	F	G	H
I	a	b	c	d				
J	e	f	g	h				
K	i	j	k	l				
L	m	n	o	p				

Figure 6 - 4x4 Luma Block

Figure 7a shows how the predictors 'a-p' are calculated from the samples 'A-M' in each of the nine modes and Figure 7b shows how the entire 16x16 block is predicted in a single operation in each of the four modes.

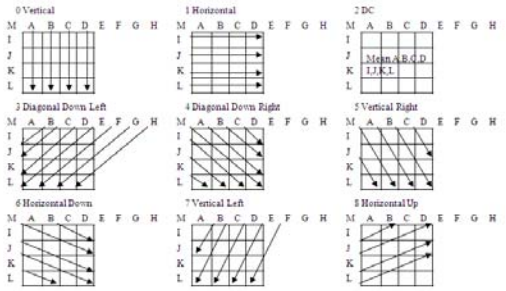

Figure 7a - 4x4 Modes

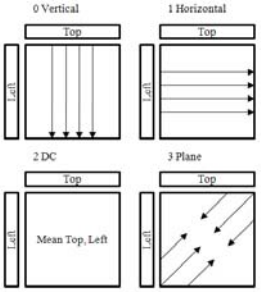

Figure 7b - 16x16 Modes

4x4 Modes:

0. A, B, C & D are extrapolated vertically.

1. I, J, K & L are extrapolated horizontally.

2. DC Mode: One value is used for all predictors, which is the mean value of A, B, C, D, I, J, K, L.

3-8. Predictor value is a weighted average of the samples depending on the direction of the arrows following the pattern: $(W + X + Y + Z + Round\_Value) >> Shift\_Value$

16x16 Modes:

0-2. The predictors are generated similar to the 4x4 modes.

3. Plane Mode: A linear function is fitted to the horizontal and vertical samples.

Let p[0,0] & p[15,15]be the top left and bottom right positions in the MB respectively.

$$pred[y,x] = Clip((a + b(x-7) + c(y-7)) >> 5)$$

$$a = 16(p[-1,15] + p[15,-1])$$

$$b = (5H + 32) >> 6$$

$$c = (5V + 32) >> 6$$

$$H = \sum_{x'=0}^{7} (x'+1)(p[-1,8+x'] - p[-1,6-x'])$$

$$V = \sum_{y'=0}^{7} (y'+1)(p[8+y',-1] - p[6-y',-1])$$

In order to implement an intra prediction module in RVC-CAL which generates predictors for all the modes, a reconfigurable processing element (PE) is used based on the architecture in [7]. Using reconfigurable PEs achieves better area usage than using a PE for each mode. A four-parallel architecture is used, meaning that there are four PEs that generate four predicted pixels in a single prediction iteration. A PE operates in a certain way depending on the prediction mode.

The module consists of a single controller and four identical series of adders, registers and shifters, each representing one of the four PEs. Figure 8 shows the RVC-CAL network of the Intra prediction engine that corresponds to the architecture in [7].

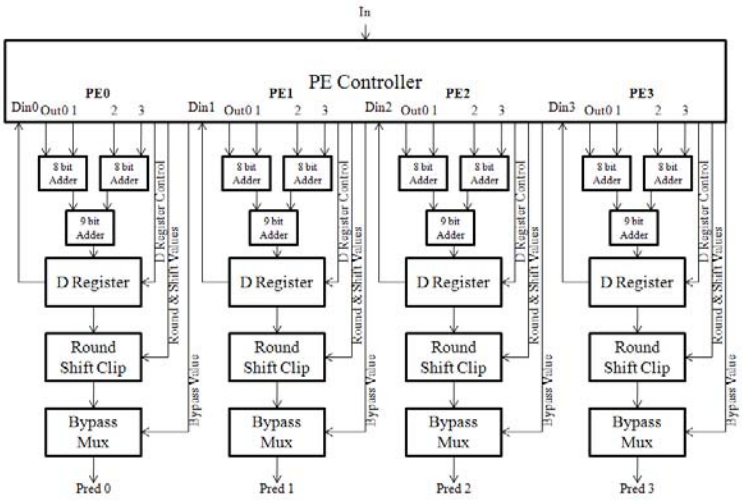

Figure 8 - Intra Prediction Module RVC-CAL Network

The main actor in the network is the *PE Controller*, which determines which values to output to the adders or to the bypass line, depending on the current mode and the position of the sample being predicted. The *8 & 9 Bit Adders* add 8 & 9 bit values. The *D Register* directs the output either to the *Round Shift Clip* actor or back to the *PE Controller* (in case addition results need to be accumulated over different prediction iterations). The *Round Shift Clip* rounds and shifts the input value and then clips it to be between 0 and 255. The *Bypass Multiplexer* outputs the predictor value either from the bypass line or the *Round Shift Clip* actor.

The operation of the intra prediction module begins with the *PE Controller* actor receiving the previously encoded samples 'A-M' serially at its input port for the current 4x4 block. The predictors for the nine 4x4 modes are then generated for the current block. In mode 0 and 1, the *PE Controller* passes a token with the appropriate value via the bypass line to the *Bypass Multiplexer*, which produces the token with the predictor value. In mode 2, the *PE Controller* produces tokens with the values to be added in the first prediction iteration at PE1 (A, B, C and D) and PE2 (I, J, K and L). The values are added and the *D Registers* are signaled by the select line to accumulate the result of the additions. In the next prediction iteration, the *PE Controller* outputs the accumulated values received at the D input ports to the adders. The final added result is shifted by 3 by the *Round Shift Clip* actor to obtain the mean value and then output at PE0. In modes 3-8, the *PE Controller* produces tokens depending on the position of the sample being predicted and the mode to the adders. It also produces tokens for the round and shift values to the *Round Shift Clip* actors. After the tokens are added the results are forwarded by the *D registers* to the *Round Shift Clip* actors and the results are output as the predicted values by the *Bypass Multiplexers*. This process is repeated for each of the 16 4x4 blocks in a single MB.

The next step is to produce the predictors for the four 16x16 modes. Mode 0 and1 are similar to those in the 4x4 block size. In mode 2, for the first 3 prediction iterations the following samples are added at each PE (Top and left samples are referred to as T0-T15 and L0-L15 respectively):

$$PE0 : (((T0 + T4 + T8 + T12) + L0 + L4 + L8) + L12)$$
$$PE1 : (((T1 + T5 + T9 + T13) + L1 + L5 + L9) + L13)$$
$$PE2 : (((T2 + T6 + T10 + T14) + L2 + L6 + L10) + L14)$$
$$PE3 : (((T3 + T7 + T11 + T15) + L3 + L7 + L11) + L15)$$

At the fourth prediction iteration, the result at each PE is added at PE0 to produce the final DC value. In mode 3, four seed values are first calculated, pred[0,0], pred[0,4], pred[0,8] and pred[0,12].

$$pred[0,0] = (A0) >> 5 \qquad pred[1,0] = (A0 + c) >> 5$$
$$pred[0,1] = (A0 + b) >> 5 \qquad pred[1,1] = ((A0 + b) + c) >> 5$$
$$pred[0,2] = (A0 + 2b) >> 5 \qquad pred[1,2] = ((A0 + 2b) + c) >> 5$$
$$pred[0,3] = (A0 + 3b) >> 5 \qquad pred[1,3] = ((A0 + 3b) + c) >> 5$$

As shown in the equations, PEs are used to calculate pred[0,1], pred[0,2] and pred[0,3] by adding b, 2b & 3b to pred[0,0] respectively. The rest of the first row is calculated in a similar fashion from the other seed values. To calculate the predictors of the second row, the first row is added to c, and so on for the rest of the rows.

The top samples are first received by the *PE Controller* and stored in state variables. After that, the predictors are generated for mode 0 similar to the 4x4 mode. The additions of the first prediction iteration in mode 2 are also performed since they need the top samples and the results of the additions are stored in the *D registers*. Next, the left samples are received by the *PE Controller* and stored in state variables. The predictors for mode 1 are then generated similar to the 4x4 mode. After mode 1 is finished, the resulting additions in the first prediction iteration of mode 2 are accumulated using the *D registers* and the remaining iterations of additions are performed, after which the final DC value is available at PE0. The *PE Controller* then receives the seed values for the plane mode and stores them in state variables. Finally, the predictors for plane mode are calculated by adding b to the seed values, in the first row and adding c to each row to calculate the values for the next row, using the *D registers* to accumulate the addition.

The *PE Controller* is the largest and most important actor in the intra prediction module. It is responsible for making the decision of which previously coded samples need to be used to generate the current predictor. It is divided into several actions, more or less one for each intra prediction mode. For Example, the actions for modes 3-8 simply produce the appropriate values to the adders depending on the pixel position (determined by the row variable) and the round and shift values. Since four predictors can be produced per prediction iteration, one row of the 4x4 block is predicted each iteration and the whole block takes four iterations to be predicted. The sequence of flow from one mode to the next, described previously, is determined by a state machine. The execution processes and the organization of the actions within the same actor are usually handled using state machines that are very similar to (yet more intuitive than) their counterparts in other HDLs.

4. Entropy Coding

The baseline profile of H.264/AVC uses two main schemes to perform high data compression: Exp-Golomb coding and Context Adaptive Variable Length Coding (CAVLC). On the higher levels (frames, etc.), H.264/AVC encodes the syntax elements using fixed or variable length codes. On the lower levels as in the slices level or below (MBs, blocks, etc.) the syntax elements are encoded using Variable Length Codes (VLC) [8].

Exp-Golomb encodes all syntax elements except for the quantized transform coefficients (residual data), which are encoded using the CAVLC scheme. Thus, the same VLC tables are used for almost all the syntax elements; which contributes to reducing the memory requirements needed to store such tables.

Exp-Golomb codes are variable length binary codes that are constructed systematically with the following pattern: $Code = [Mzeros][1][INFO]$

The code words constructed in this fashion are guaranteed to have symmetric width; where the INFO field is represented in M bits, making the width of the code word equal to $2M+1$.

Given a parameter k, the corresponding *code_num* is then calculated according to one of four modes and the mode selection decision is based on the parameter type. Each of such modes is designed to produce shorter codewords for frequently-occurring values and longer codewords for less common parameter values. After the *code_num* has been calculated, codewords can be constructed on the basis of the following equations.

$$M = \lfloor \log_2 (Code\_Num + 1) \rfloor \qquad INFO = Code\_Num + 1 - 2^M$$

Figure 9 [3] shows the Exp-Golomb module implemented in RVC-CAL as a simple network with one input port, which receives the parameter token to be encoded and one output port which outputs the codeword serially. The Exp-Golomb network provides interconnections among nine different actors, out of which four (plus one, on which more later) actors perform the tasks of the four different mapping modes. The rest of the actors include a *controller* responsible for the mapping mode decision based on the parameter type, a *code generator* to construct the Exp-Golomb codeword as described above, an *assembler* responsible for the reordering, concatenation and outputting of the codeword bits, and finally a *utility* actor that provides decimal to binary conversion to represent the INFO bits.

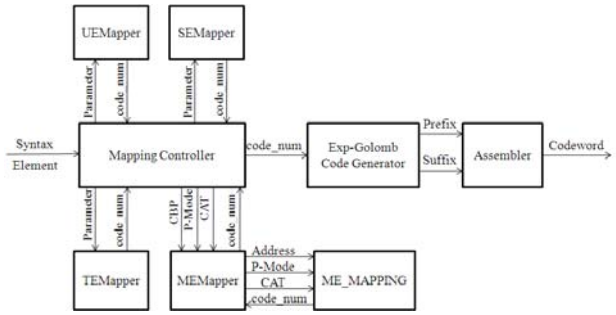

Figure 9 - Exponential Golomb RVC-CAL Network

For each of the four different mapping modes an actor specifies how the mapping from the parameter to the *code_num* is to be performed. The *unsigned mapping* actor is a simple one, which outputs the *code_num* token with a value equal to that of the input token. The *signed mapping* actor defines two different actions depending on the value of the input token; one in which the *code_num = 2|k|* for non-negative values and the other in which the *code_num = 2|k|-1* for negative values. The *truncated mapping* actor also defines two different actions following the unsigned mapping scheme for values greater than one and inverting the binary value of the input token otherwise. The mapped exponent relies on a lookup table (LUT) specified in the ITI-recommendation based on different prediction modes and chroma array types. Thus, the *mapped exponent* actor is a bit more involved than the other three. It acts as a controller communicating with a dedicated actor storing the LUT (targeting a ROM implementation upon synthesis). The communication channel is interfaced with the corresponding address (input) and data (output) ports. The LUT is organized in a reordered manner, so that the *coded_block_pattern* is used as an address for the table to read back the corresponding *code_num.*

Implementing the Exp-Golomb module in RVC-CAL provides a high degree of abstraction that allows for seamless integration within other modules with minimal effort. As opposed to other implementations provided in C for example, no pre-knowledge of the user defined data types or any other implementation-specific details are needed. Integrating the Exp-Golomb module within the larger entropy coding module is as easy as defining the interconnections for its input and output ports; that adds only two lines of code. Thus, the Exp-Golomb module and the CAVLC module can be implemented by two completely different developers and then the integration overhead almost sums up to zero. Table 1 holds comparisons related to development time and lines of code of the proposed approach against traditional approaches.

The H.264/AVC recommendation [9] provides two alternative entropy coding methods (both of which are context-adaptive variable-length based), Context Adaptive Variable Length Coding (CAVLC) and Context Adaptive Binary Arithmetic Coding (CABAC). Context based adaptivity improves the performance considerably relative to prior standards. Since CABAC is not a part of the baseline profile only CAVLC is considered hereby.

CAVLC exploits the statistical properties of the quantized 4x4 block with the coefficients to be encoded to provide a compact and efficient lossless representation of the data. It is also context adaptive in the sense that different VLC tables are used for the different syntax elements and are switched according to the values of previously coded elements [8]. Entropy coding performance is improved in comparison to other single-table based schemes because the different VLC tables are designed to accommodate the specific statistics of each syntax element.

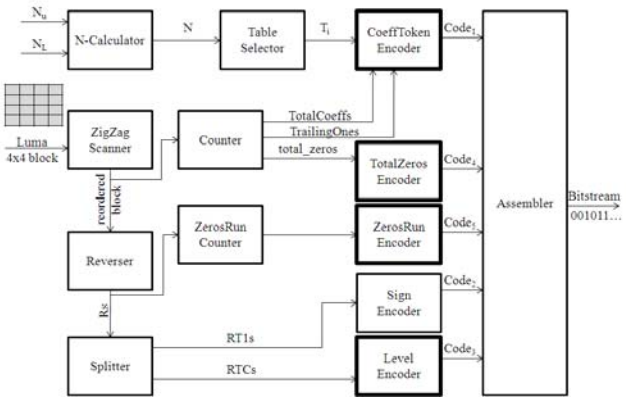

Figure 10 - CAVLC RVC-CAL Network

Figure 10 [3] describes the proposed RVC-CAL implementation of the CAVLC algorithm (thoroughly described in [5]). It starts with the *Zigzag Scanner* actor performing the pre-processing on the block of coefficients and making the reordered block available for both the *Counter* and the *Reverser*. The *Counter* prepares the meta-data needed throughout the algorithm execution; namely the *TotalCoeffs*, *TrailingOnes* and the *total_zeors*. The *N Calculator* calculates the number of non-zero coefficients based on the corresponding values of the neighboring blocks. The *CoeffTokenEncoder* uses the *TotalCoeffs* and *TrailingOnes* to access one of the LUT to retrieve the corresponding codeword. The choice of the LUT to be accessed is made by the conjunction of the *N-Calculator* and the *Table Selector* actors. The algorithm execution proceeds in a distributed fashion among the other actors and follows naturally as illustrated in Figure 10.

The algorithm doesn't execute according to its intuitive order but rather depending on the tokens available at each point in time during execution and that adds yet another advantage to the RVC-CAL implementation. The different actors in Figure 10 execute in independently and it is then the responsibility of the *assembler* to compile the output tokens from the different actors, reorder them and output the encoded stream serially.

H.264/AVC CAVLC encoding relies heavily on the usage of LUTs (Figure 10 shows the sub-modules in which lookup is involved having a thicker border), something which provides significant improvement of efficiency but with the price of complication of the fabrication process and additional consumption of area. The proposed CAVLC module introduces a memory model which preserves the complete LUTs nevertheless still sparing up to 31.5% of the area required to store them.

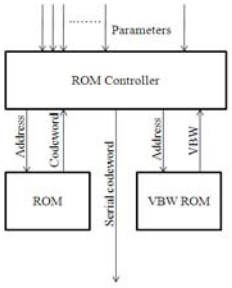

Figure 11 – CAVLC LUT Memory Model

Figure 11 shows the proposed memory model which is represented in RVC-CAL as an actor representing the memory controller, another representing the memory itself and a third with the valid bit widths (VBW) of the corresponding entries of the second actor. This approach exploits an efficient storage technique for the run of zeros to the left of the codeword. For example the codeword 00000000001 is stored as only 001 in the memory with an 11 in the corresponding location in the VBW memory. In this example, only 7 bits are to be stored instead of 11. With almost 475 different variable length code words to be stored [9], such reduction multiplies and offers approximately 21% reduction in the ROM usage. It is then the responsibility of the controller to align the codeword before outputting it, a task which requires minimal computational interference.

5. Results and Analysis

The modules described in this paper are the main modules that were integrated along with others to form an AVC baseline encoder. Figure 12 shows the subjective results of reconstructing a frame, encoded using different Quantization Parameters (QP).

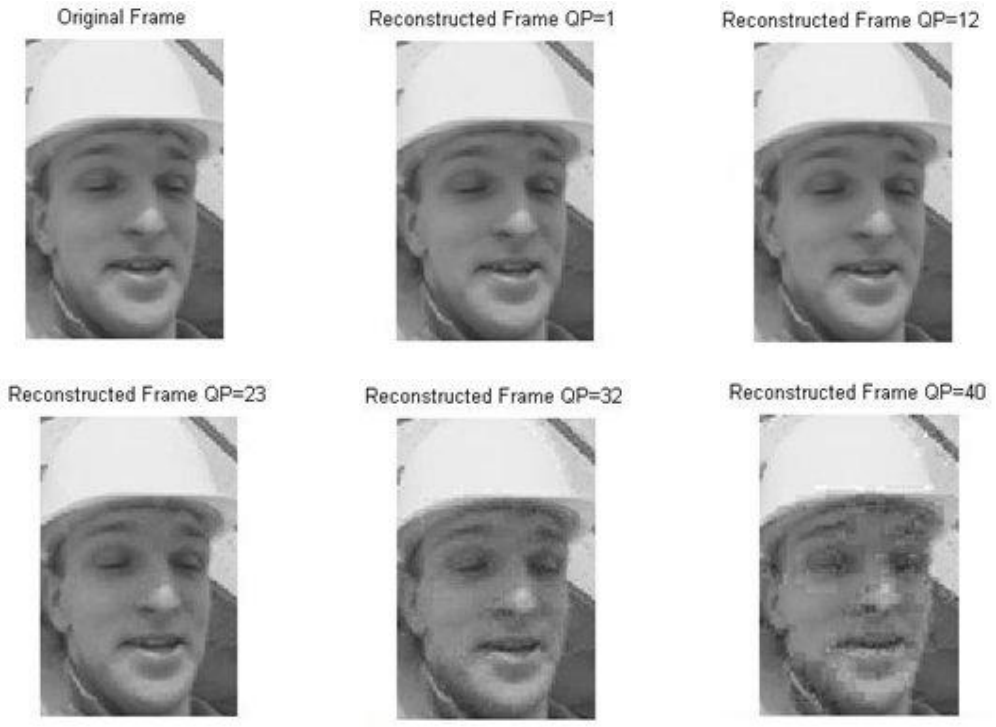

Figure 12 - A close-up on the first reconstructed P-frame of Foreman QCIF

The feasibility of choosing RVC-CAL as the implementation language is reflected on the results of comparing the encoded versus the reconstructed frame. Figure 13 shows the Peak Signal to Noise Ratio (PSNR) of the reconstructed frame measured at different QPs.

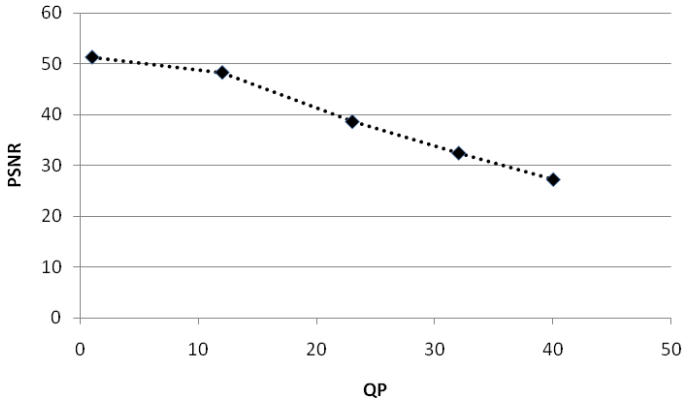

Figure 123 - PSNR at different QPs

One of the major RVC advantages is that it accompanies its normative description language (RVC-CAL) with many supporting tools that enable automatic code generation into software (CAL2C) and hardware (CAL2HDL) [10]. Table 1 presents a comparison between the proposed RVC-CAL implementation, the H.264/AVC JM reference software written in C [11] and a reference VHDL implementation.

Table 1 - Comparison between RVC-CAL, C and VHDL approaches

	Lines of Code (LOC)			Development Time (MH)			Number of Developers		
	Inter[1]	Intra	Entropy	Inter	Intra	Entropy	Inter	Intra	Entropy
RVC-CAL	203	1239	922	56	80	72	1	1	1
C/C++	356	2758	1765	N/A*			3	5	3
VHDL	897	N/A*	3784	133	N/A*	116	1	N/A*	1

[1] The comparison relates to integer pel inter prediction implementing full search and SAD.
* No precise data available at the time of comparison.

Table 1 [3] shows the lines of code, development time and number of developers required for the RVC-CAL, C and VHDL implementations correspondingly. The numbers show that the RVC-CAL implementation needs less time to be developed and hence requires fewer developers. This gives the RVC-CAL implementation an advantage of reducing the development costs, while at the same time minimizing the TTM.

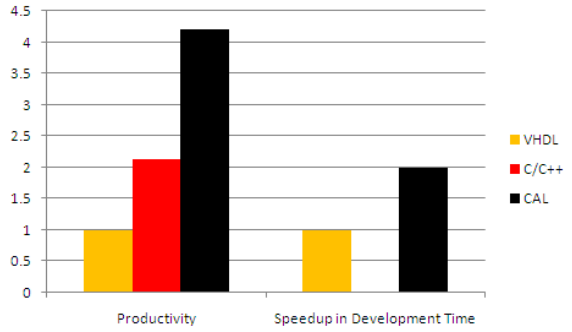

Figure 134 - The Gain from Using the RVC-CAL Implementation

Figure **13**14 shows the average productivity gain (calculated as the average saving in LOC) and the development speedup achieved from using the proposed RVC-CAL approach. Both metrics are normalized with respect to the VHDL implementation to focus on the improvement factor. The proposed RVC-CAL implementation for the encoder is four times more productive than the corresponding VHDL implementation, although it required only half the time. On the other side, RVC-CAL is also twice as productive as the C/C++ implementation. Enhancements to RVC-CAL currently taking place propose embedding built-in functions that are expected to increase the RVC-CAL productivity up to the double.

The results achieved are attributed to a number of factors referring to the strong abstraction and encapsulation properties exhibited by CAL. Such properties allow the developers to focus on one module at a time, without worrying about the order of execution, the synchronization between the different modules or any other process-irrelevant details.

On the other hand, the results are echoed on the hardware implementation level. The HDL model is generated from the presented CAL model using the CAL2HDL tool. The HDL code is synthesized using Xilinx ISE targeting the Xilinx Virtex 5 XC5VLX50T FPGA. The synthesis results of the CAVLC module are provided as an example for the quality of the hardware implementation. Table 2 summarizes the performance of the CAVLC module and compares it against [8] and [12]-[15].

Table 2 - Performance of the CAVLC module compared to other implementations

	Critical Path (ns)	CLK Frequency (MHz)	Number of LUT	Throughput (MSamples/s)
Proposed Implementation	3.729	268.1	112	268
[8]	9.6	103.8	2,467	103.8
[12]	31.326	31.9	84,902	510.4
[13]	3.1	210	N/A*	100
[14]	8	125		74.04
[15]	13.15	76	3,946	6.75

* No precise data available at the time of comparison.

The synthesis results show that the proposed implementation exceeds the throughput of the others with a factor of 2.58 in the worst case, with the exception of [12] because that implementation employs hardware redundancy to exploit parallelism and consumes 758 times more hardware resources. Besides the advancements in FPGA manufacturing technology, the results can be attributed to several factors. The abstract and encapsulated implementation facilitated by RVC-CAL allows for optimization of every sub-module (actor) on its own, which collaborates to deliver overall optimization of the whole CAVLC module. In addition, the optimizations performed by the CAL2HDL tool during the HDL code generation (explicitly specified in [16]) also contribute greatly to the quality of the synthesis results.

6. Conclusion

In this paper, an RVC-CAL implementation of the major FUs constituting an H.264/AVC baseline profile encoder has been featured. Throughout the different sections, a brief overview of the RVC-CAL approach along with the associated networks has been presented for each of the inter, intra frame predictors, and the entropy coder. The components presented are potential candidates for addition to the VTL of the RVC standard, due to the abstract and encapsulated representation that was made feasible by RVC-CAL.

The results illustrate the advantages of using RVC-CAL as a specification language for the RVC standard. RVC-CAL implementation can target both software and hardware architectures; unlike the reference software written in C (sequential programming language) or VHDL (hardware description language). Development in RVC-CAL needs less time and the code produced by the generators is more efficient than the corresponding code in C or VHDL. The results in table 1 are also reflected in [16] where an MPEG-4 simple profile decoder was developed using RVC-CAL and compared to a VHDL implementation. Not only was the RVC-CAL development time four times faster, but it was also more efficient in terms of execution time. This was attributed mainly to the advantages of using dataflow programming instead of register transfer level (RTL) design. Furthermore, the faster RVC-CAL implementation time allowed for more design optimization cycles. It is generally considered that using a lower level language such as VHDL for hardware implementation would produce better results than a higher level language implementation such as RVC-CAL due to the ability to control more implementation details. However, using RVC-CAL allows for faster design cycles, thus providing regular feedback to designers and enabling them to optimize the design faster and more frequently.

RVC-CAL allows developers to focus on functional issues thus leaving the timing details to the code generators which also perform optimizations to adjust the clock rate [16]. The code generators also handle the handshaking signals required to simulate the connections between the actors and networks so the simplicity of using tokens for communication in the CAL specification does not suffer from any overhead when translating to hardware description language. This stems from the properties of dataflow programming. Dataflow programs are naturally concurrent [16] and they are easily reused and reconfigured, which is an important aspect of the RVC framework. Furthermore, simulating parallelism is another advantage to using data flow programming over sequential programming that makes development faster [17]. In C, parallelism must be explicitly defined, which is often difficult and adds significant overhead. This also hides the original structure of the program among the code required to handle threads and platform specifics. However, in CAL parallelism is implicit; actions are fired and produce and consume tokens

whenever the firing conditions are satisfied. Normally, developers need not pay attention to the order of action execution.

References

[1] Jang, E. S., Ohm, J., Mattavelli, M. (January 2008). Whitepaper on Reconfigurable Video Coding (RVC). ISO/IEC JTC1/SC29/WG11 document N9586.

[2] Lucarz, C., Mattavelli, M., Thomas-Kerr, J., Janneck, J. (2007). Reconfigurable Media Coding: A New Specification Model for Multimedia Coders. Workshop on Signal Processing Systems.

[3] Aman-Allah, H., Hanna, E., Maarouf, K., Amer, I. (2009). Towards a Comprehensive RVC VTL: A CAL Description of an Efficient AVC Baseline Encoder. IEEE International Conference on Image Processing (ICIP 2009).

[4] Mattavelli, M. (2008). Reconfigurable Video Coding (RVC) a new Specification and Implementation Paradigm for MPEG Codecs. The 12th Annual IEEE International Symposium on Consumer Electronics. [Keynote Presentation].

[5] Richardson, I. E. G. (2003). H.264 and MPEG-4 Video Compression. Aberdeen. Wiley.

[6] Yang, W. (2003). An Efficient Motion Estimation Method for MPEG-4 Video Encoder. IEEE Transactions on Consumer Electronics, 49(2).

[7] Huang, Y., Hsieh, B., Tung-Chien C., Chen, L. (2005). Analysis, Fast Algorithm, and VLSI Architecture Design for H.264/AVC Intra Frame Coder. IEE Transactions on Circuits and systems for Video Technology, 15(3), 378-401.

[8] Silva, T., Vortmann, J., Agostini, L., Bampi, S., Susin, A., (2007). FPGA Based Design of CAVLC and Exp-Golomb Coders for H.264/AVC Baseline Entropy Coding. 3rd Southern Conference on Programmable Logic (SPL07).

[9] International Telecommunication Union. (2005). Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Recommendation H.264 (03/05); Advanced Video Coding for Generic Audiovisual Services).

[10] Lucarz, C., Mattavelli, M., Wipliez, M., Roquier, G., Raulet, M., Janneck, J., et al. (2008). Dataflow/Actor-Oriented language for the design of complex signal processing systems. Conference on Design and Architectures for Signal and Image Processing (DASIP 2008).

[11] Joint Video Team (JVT) reference software, version 14.2. [Online] http://iphome.hhi.de/suehring/tml/download/old_jm/jm14.2.zip.

[12] Amer, I., Badawy, W., Jullien, G. (2004). Towards MPEG-4 Part 10 System on Chip: A VLSI Prototype for Context Based Adaptive Variable Length Coding (CAVLC). IEEE Workshop on Signal Processing Systems, p. 275-279.

[13] Yi, Y., Cheol Song, B., (2008) A Novel CAVLC Architecture for H.264 Video Encoding At High Bit-rate. IEEE International Symposium on Circuits and Systems (ISCAS 2008).

[14] Chien, C., Lu, K., Shih, Y., Guo, J. (2006). A High Performance CAVLC Encoder Design for MPEG-4 AVC/H.264 Video Coding Applications. IEEE International Symposium on Circuits and Systems, p. 3838-3841. (ISCAS 2006).

[15] Sahin, E., Hamzaoglu, I. (2005). A High Performance and Low Power Hardware Architecture for H.264 CAVLC Algorithm. 13th European Signal Processing Conference. (EUSIPCO 2005).

[16] Janneck, J., Miller, I. D., Parlour, D. B., Roquier, G., Wipliez, M., Raulet, M. (2008). Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. IEEE Workshop on Signal Processing Systems (SiPS 2008).

[17] Bhattacharyya, S., Brebner, G., Janneck, J., Eker, J., von Platen, C., Mattavelli, M., Raulet, M. (2008). OpenDF – A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems. First Swedish Workshop on Multi-Core Computing.