A unified hardware/software co-synthesis solution for signal processing systems

Endri Bezati, Hervé Yviquel, Mickaël Raulet, Marco Mattavelli

▶ To cite this version:

HAL Id: hal-00717244
https://hal.archives-ouvertes.fr/hal-00717244

Submitted on 12 Jul 2012

A UNIFIED HARDWARE/SOFTWARE CO-SYNTHESIS SOLUTION FOR SIGNAL PROCESSING SYSTEMS

Endri Bezati[1], *Hervé Yviquel*[2], *Mickaël Raulet*[3], *Marco Mattavelli*[1]

[1] Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland
{firstname.lastname}@epfl.ch
[2] IRISA/University of Rennes 1, F-22300 Lannion, France
{firstname.lastname}@irisa.fr
[3] IETR/INSA of Rennes, F-35708 Rennes, France
{firstname.lastname}@insa-rennes.fr

ABSTRACT

This paper presents a methodology to specify from a high-level data-flow description an application for both hardware and software synthesis. Firstly, an introduction to RVC-CAL data-flow programming and Orcc framework is presented. Furthermore, an analysis of a close to gate intermediate representation (XLIM) is bestowed. As a proof of concept a JPEG codec was written purely in RVC-CAL to test the co-synthesis tools and then an analysis of the generated hardware and software results are given. Our experience shows that using RVC-CAL can unify the process of creating the same application for software and hardware without modifying a single source code for each solution.

Index Terms— Co-Design, Co-Synthesis, Dataflow, FPGA, JPEG, OpenForge, Orcc, RVC-CAL, XLIM

1. INTRODUCTION

Signal processing applications becomes more and more complicated and their complexity continuously grow at each generation. This is the case, for instance, of video compression standards that following the demand for higher quality video transmitted by smaller and smaller bandwidths, achieve the objective at the expense of introducing, at each new standard release, a large increase of codec complexity. Developing implementations for heterogeneous platform of such applications is always a difficult challenge. Currently frameworks capable of generating code from the same, ideally high-level specification, for both Hardware and Software synthesis are not available or presents severe limitations.

C is one possible high level language as C to Gates tools and their corresponding design flows (ImpulseC [2], Handel-C [10] and Spark [7]) generate VHDL code from C-like spec-

ifications. Thus, the entire design space is far from being completely explored because these tools handle only hardware code generation. Moreover, approaches to high-level hardware synthesis fall broadly into two categories:

- those that attempt to adapt software programming languages to the creation of hardware by creating tools that translate software programs into circuit descriptions such as Catapult C, c2h, PICO Express, ImpulseC,

- those that devise one or more new languages (textual or visual), designed to be more amenable to the generation of efficient hardware e.g., Handel-C [5], Mitrion-C, Mobius.

The approaches in the first category attempt to leverage software tools and a large community of programmers. However, the goal of translating real-world applications written in a language such as C into efficient hardware implementations has proven elusive, despite considerable efforts in this direction.

Although hardware code generation by the CAL data-flow language has been presented in the past [3, 9] with the OpenDF framework, this paper presents an approach for unified hardware and software synthesis starting from the same program (specification).

This paper is organized as follow: Firstly, Section 2 gives a brief introduction of CAL data-flow programming and its associate compiler called Open RVC-CAL Compiler. Then, two sections make the following contributions:

- We present a complete compilation flow from RVC-CAL towards HDL synthesis (Section 3). This flow uses the XLIM Intermediate Representation (IR), an XML format for representing a language independent model of imperative programs based on a well-known form called three-address code (TAC or 3AC).

- We give a co-design case study, in which a JPEG Codec written only in RVC-CAL is partitioned into components and then synthesized to both SW and HW (Section 4).

Finally Section 6 outlines the current limitation of the approach and discusses the perspectives of future extensions.

2. BACKGROUND

This section presents RVC-CAL, a standardized subset of the original CAL Actor Language, the Open RVC-CAL Compiler, an *open-source framework* that supports RVC-CAL for generating implementation code, and OpenForge, a synthesis tool developed by Xilinx.

2.1. RVC-CAL Data-flow Programming

CAL Actor Language is a language based on the Actor model of computation for data-flow systems [6]. An actor, is a modular component that encapsulates its own state. Each actor interacts with each other through FIFO channels, see Figure 1. An actor in general may contain state variables, global parameters, actions, procedures, functions and finite state machine that control the executions of actions. CAL enables concurrent development and provides strong encapsulation properties. CAL is used in a variety of applications and has been compiled to hardware and software implementations. The RVC-CAL language is a subset of the CAL language and it is normalized by ISO/IEC as a part of the RVC standard. Although it has some restrictions in data types and features that are in used in CAL [4, 6], is sufficient and efficient for specifying streaming and signal processing systems such as MPEG compression technology.
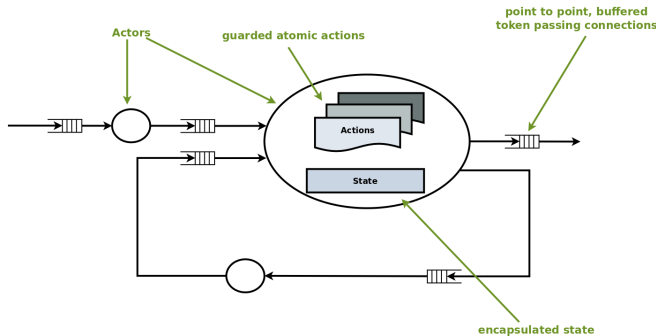

Fig. 1. The CAL computing model.

2.2. Open RVC-CAL Compiler

The Open RVC-CAL Compiler (Orcc)[1] is an open-source framework designed to generate implementation code from a network of RVC-CAL actors specified by a network topology description [15].

The **Frontend** has the task of parsing all actors and translating them to an Intermediate Representation (IR). Such IR is a data structure that is built from input data (the actor) to
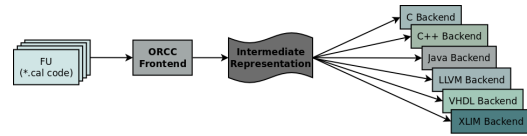

Fig. 2. Orcc framework chain.

a program, and from which part or all of the output data of the program is constructed in turn. The next steps is to run a language target-specific back-end. The idea is to let flexibility to the backends for generating optimized code and to preserve features of the CAL language that does not overspecify scheduling information.

The **Backends** generate code depending on the targets. Their purpose is to create target specific code. Each backend will parse the hierarchical network from a top-level network and its child network. Also optionally it flattens the hierarchical network. Orcc for the moment offers a variety of back-ends. These back-ends are C, C++, LLVM, VHDL, XLIM, etc.

To generate a software decoding solution we used the C backend of Orcc. The generated C code is ANSI-C compatible and it is portable to different platforms such as Windows, Linux, Mac OS X and others.

Orcc gives the possibility to create native actors and native procedures. A native actor can be written directly in C or VHDL (for modelsim simulation). The purpose of these native actors is to offer the possibility to use the host Input/Output (write a file, display an image). As RVC-CAL has not its standard library to communicate with the host, native actors permit that.

2.3. OpenForge and HDL code generation

Forge was a research tool developed by Xilinx for their C to gate implementation. Forge has fallen to the public domain and was renamed to OpenForge. The first tool that permitted the CAL to HDL was the OpenDF framework (a CAL simulator) with the use of OpenForge as a backend for the Verilog HDL generation. Often in RVC-CAL literature this tool is called as CAL2HDL [9, 12]. XLIM OpenDF code generation does not support all the RVC-CAL subset and it is too slow compared to the XLIM generation of the Orcc. Thus the choice of creating an XLIM backend was a must.

OpenForge takes as an input an XLIM file, which is just the representation of the static single assignment (SSA) form of an actor in an XML format. Then the synthesis stage follows with the analysis of the SSA representation into a web of circuits built from a set of basic operations like arithmetic, logic, flow control, memory accesses and etc.

Also OpenForge as an intelligent synthesis tool supports the unrolling of loops, or the insertion of registers to improve the maximal clock rate of the general circuit (pipe-lining). Finally the OpenForge will generate a Verilog file that repre-

sents the RVC-CAL actor with an asynchronous handshake-style interface for each of its ports. Orcc generates a Top VHDL that connects the Verilog generated actors with back-to-back or using FIFO buffers into a complete system. Also the FIFO buffers can be synchronous or asynchronous (given the user choice). Give the previous statement it is easy to support multi-clock-domain data-flow designs (different clock domains can be given directly from the Orcc user interface).

Orcc can also generate directly VHDL code for the actors and the network [13], but as the VHDL code generation is not mature enough it was not used for this implementation. Future work on the VHDL Backend will maybe provide an alternative to the OpenForge.

3. XLIM CODE GENERATION

This section presents first the XLIM representation then the compilation process of an RVC-CAL application towards a hardware target.

3.1. Presentation of the XLIM representation

The XML Language-Independent Model (XLIM) is an intermediate representation (IR) developed by Xilinx [1] to make the code optimizations of data-flow programs easier. Indeed, the source code written by developers and also the abstract syntax tree (AST) usually produced by the parser at the beginning of the compilation process are not ideal to compute analysis and transformation of code due to their lexical structure.

XLIM is mainly an XML document containing several elements which describes the behavior of a data-flow actor: the interfaces of the actor (inputs and outputs), the set of state variables, the computational procedure of each action and finally the action scheduler which manages the execution of the actions according to.

The XLIM representation is a close to gate representation which requires the respect of the following properties to represent directly the dependency relation between the different elements of the program and consequently permit code optimizations for hardware targets:

- **Static single assignment form (SSA)** requires that each variable used by the program is assigned exactly once. As a consequence, a variable assigned several times in the initial form of IR is transformed in different versions of the variable (for example a variable x which is assigned three times is transformed on three variables x_1, x_2 and x_3) and some special statements called ϕ-functions are inserted at join nodes of the control-flow graph in order to assign a variable according to the execution path (for example the statement $x_3 \leftarrow \phi(x_1, x_2)$ expresses that x_3 has the value of x_1 if the program jumped from the first node and x_2 if it jump from the second one).

- **Three-address code (3AC)** representation describes each basic operation executed in the program (like addition or multiplication) by the following 4-tuple $(Operator, Operand1, Operand2, Result)$ where $Operand1$, $Operand2$ and $Result$ are the variables and $Operator$ is a primitive operator (an arithmetic operator for example). As a result, there is no more complex expression containing several primitive operations.
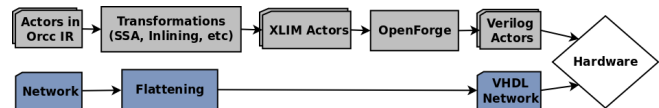
3.2. XLIM backend

Fig. 3. The compilation flow of the XLIM backend.

The Open RVC-CAL Compiler (Orcc) includes an XLIM backend which corresponds to an RVC-CAL frontend for tools like OpenForge or XLIM2C [14]. The default compilation flow is presented in Fig. 3 and consists on several passes and particularly some transformations the intermediate representation of Orcc:

- **Inlining** of RVC-CAL functions and procedures, indeed the XLIM does not support the call instruction because of its low representation level.

- **SSA transformation** is made to validate the needed SSA property and consists in indexing variables and adding ϕ-functions.

- **3AC transformation** splits complex expressions including several primitive operations to multiple 3AC compliant instructions.

- **Copy propagation** is an optimization which removes the direct assignment of a variable to another variable like $a := b$ by replacing all uses of a by b.

- **Cast adder:** the XLIM representation use a precise type system (for typename and size) which needs explicit cast instructions. These cast instructions are added thanks to the bit exact precision of Orcc IR by visiting the type of each expression and variable.

- **Array flattener** transforms multidimensional arrays to unidimensional ones and made computation of right index.

After these transformations, the actors are printed in XML format respecting XLIM properties using a template engine called *StringTemplate* [11]. This mechanism permits to quickly generate XLIM files (at most few seconds), increases flexibility (several backends were developed in Orcc) and reduces maintenance cost (a template is easier to change than a program).
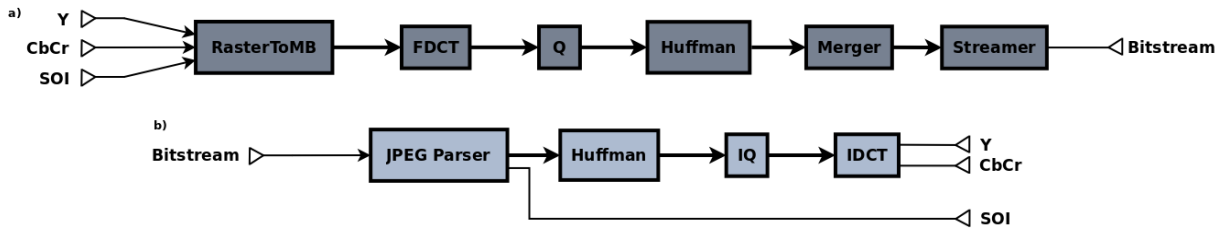
Fig. 4. The RVC-CAL Codec: a) JPEG Encoder, b) JPEG Decoder.

4. USER CASE: AN RVC-CAL JPEG CODEC

As a user case a JPEG Codec was chosen and it was written purely in RVC-CAL[2]. The JPEG codec is based on the ITU-T. IS 1091 standard. The idea was to implement the encoder in the FPGA and the decoder in a computer host. The JPEG Codec is implemented based on the simple profile and is using a static quantification and Huffman Table. This was chosen only for simplicity, using RVC-CAL is very easy to add an actor that can have as inputs different quantification and/or Huffman tables without changing the JPEG codec model structure.

4.1. RVC-CAL JPEG Encoder

The RVC-CAL JPEG encoder is modeled as a serial data-flow application. Then encoding is done at a Macro-block (MB) level. The input of the encoder is giving in Raster 4:2:0 YUV format (see Fig.5) , this format was chosen due to the output of the input camera. The encoder is separated in six actors. The JPEG standard describe the encoding in a block level of 64 pixels. In 4:2:0 format there are four luminance (Y) blocks of 8x8 for two chrominance (one for U and one for V) blocks of 8x8.
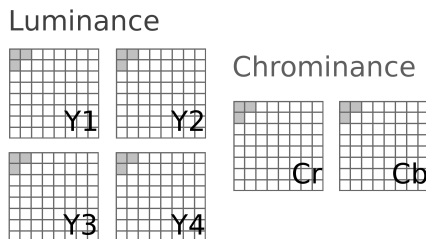

Fig. 5. The YUV 4:2:0 Macro-Block representation.

So the first actor in the encoder is the Raster to MB operation. The actor is taking as inputs the Y, Cb and Cr together and a signal SOI that indicates the Size Of the Image. The next step is to transform the YUV pixels to a Forward Discrete Cosine Transform (FDCT) and to quantify them. The FDCT and the Quantization works in 8x8 block level. So for

---

[2]The JPEG Codec is available at http://orc-apps.sf.net

an MB the FDCT and the Quantization actors are processing six blocks of 8x8, this can give a potential parallelism in the JPEG algorithm, this will be described in the future work section. The most important part of the JPEG encoder is the Huffman actor. This actor is specific to 4:2:0 MB scheme, it will treat the luminance blocks and then the two chrominance blocks. The Huffman will generate two bit-streams (not shown in the Fig.4), one for the luminance and one for the chrominance. For simplifying the JPEG model a merger actor was created so that it can serialize the previous bit-streams. Finally the Streamer actor will add all the necessary information (start/stop flags, quantization and Huffman tables) so that a proper JPEG bit-stream is correctly generated.

4.2. RVC-CAL JPEG Decoder

The RVC-CAL Decoder is the inverse process of the JPEG encoder, but with quality loss of the encoded image due to the lossless nature of the JPEG compression. The decoder is separated in four actors. The first step is to decode the JPEG bit-stream. So the JPEG Parser is retrieving all the necessary information form the bit-stream. Next, the Huffman actor is responsible to decode the Huffman encoding of the transformed and quantified 4:2:0 YUV blocks. Finally the inverse quantization and the Inverse DCT (IDCT) actors form the 4:2:0 MBs of the reconstructed image.

4.3. A Co-Design example of the JPEG Codec

Orcc framwork offers the possibility to generate source code for hardware and software. Given the RVC-CAL of the JPEG Codec, the encoder can be implemented in a FPGA board and the decoder can be compiled as an ANSI C program so that it can run on each platform that has an ANSI C compiler. For our user case a Virtex 6 FPGA board with a PCI-express connector and an Intel iCore 7 PC host were used. The communication between the FPGA board and the PC was done via the PCI-Express port.

The RVC-CAL JPEG encoder is implemented in the Virtex 6 FPGA board. Orcc first generates the XLIM intermediate representation and then it calls the OpenForge Back-end.In the end the Verilog source code files compose the generated code for each actor and a VHDL file that represents the network of the actors. Xilinx PCIe IPCore does the commu-

Fig. 6. The RVC-CAL JPEG Codec partitioned in Hardware and Software.

nication between the generated code and the PCI-express. A camera connected with the FPGA board via the HD-SDI interface gives the acquisition of the input image. Then the image is compressed by the RVC-CAL JPEG Encoder and is then transmitted by the PCI-Express.

Xilinx provides a basic Linux driver for the PCI-Express port, which permits to read and write values from the PCI-Express bus. With the help of the native actors in Orcc, a read or source PCI-Express actor was written so that the RVC-CAL C or C++ generated application could communicate with the PCI-Express bus. The RVC-CAL JPEG decoder application is generated in C so that it can be implemented in the PC Host. As an input for the JPEG decoder the PCI-Express source actor is giving a correct JPEG bit-stream to decode. Then finally the decoder decodes the bit-stream and is then displaying the images on the computer screen.

4.4. Results

The RVC-CAL JPEG Codec has 1653 source code lines: 990 lines for the encoder and 663 lines for the decoder. The numbers of lines are quite comparable for those of an entirely written JPEG codec in pure C. Thus the number of lines depends on the authors source code and how he/she is programming. Hence writing in RVC-CAL is really easy and intuitive. For example the Fig. 7 presents the source code of the Quantization actor written in RVC-CAL.

The RVC-CAL JPEG encoder takes 22 % of the Virtex 6 FPGA (see Fig. 8) and it can encode 14 Frames per second with a 50Mhz clock (the time was measured by Xilinx Chip scope) for a set of 512x512 input images. The time for encoding 30 images of 512x512 plus the transfer from the PCI-express bus is $\simeq 3$ seconds.

The open-source Xilinx PCI-Express driver is too slow compared to the PCI-Express 1x standard, the images are stocked in the DDR RAM of the FPGAs board and it takes almost one second to pass the encoded images from the DDR to the host via the PCI-Express bus. Thought no optimization in RVC-CAL code was done and the serial architecture of the RVC-CAL JPEG encoder is penalizing the throughput. A simple splitting of the YUV components can increase the theoretical performance by 3 even 5 if an intelligent splitting is implied in the 4 blocks of the Y.

In the other side of the PCI-Express bus the JPEG Decoder can decode 135 Frames/sec for a set of images with a resolution at 512x512. Still here the potential parallelism of the YUV splitting it is not taken in account due to the serial ar-

```
package jpeg.encoder;

import jpeg.encoder.common.Tables.QT;
import jpeg.encoder.common.Tables.zigzag;

actor Quantization()
int(size=32) In ⇒ int(size=32) Out:
int Block_Type := 0; // Block_Type = 0,1,2,3 (Luma),
                     // Block_Type = 4,5 (Chroma)

 Quant: action In:[val] repeat 64 ⇒ Out:[data] repeat 64
 var
  List(type:int(size=24), size=64) data
 do
  foreach uint i in 0 .. 63 do
   if (val[zigzag[i]] > 0) then
    data[i] := ( val[zigzag[i]] +
               (QT[Block_Type >> 2][i] >> 1)
               ) / QT[Block_Type >> 2][i];
   else
    data[i] := ( val[zigzag[i]] −
               (QT[Block_Type >> 2][i] >> 1)
               ) / QT[Block_Type >> 2][i];
   end
  end
  Block_Type := (Block_Type + 1) mod 6;
 end
end
```

Fig. 7. The Quantization (Q) actor in the JPEG encoder written in RVC-CAL.

Logic utilization	Used	Utilization %
Registers	10033	6
Slice LUTs	13308	16
LUT-FF pairs	4241	22
IOB	85	14
Block RAM	85	6

Fig. 8. Synthesis Information on Virtex 6 FPGA

chitecture of the RVC-CAL JPEG Decoder.

5. CONCLUSION AND FUTURE WORK

This co-synthesis solution is still in progress and a lot of work is to be done. The work is separated in two fronts, the code generation and the code optimization of the RVC-CAL Applications (JPEG Codec for this paper). Code generation is naturally separated in software and hardware code generation. Different metrics and code analysis are being developed so that the generated code is more efficient. Although in software code generation the current bottleneck is the network and actor scheduling, efforts are being given for a dynamic scheduling that will take into account the dynamic execution of actors in a multi-core platform and the load balancing.

As for hardware code generation, optimization can be done directly in the XLIM code generation for reducing the number of Slices in the synthesized code. Due to the nature of the Orcc IR a lot of intermediate variables are added so that the generated code for software is more efficient and easier to
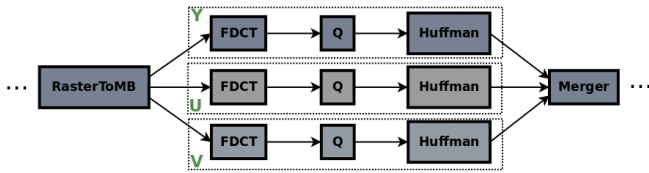
Fig. 9. Potential YUV component parallelism for the JPEG encoder.

interpret by the C/C++ compilers. But OpenForge is not recognizing that the intermediate variable comes from the same variable so it is adding more registers. A test was conducted between the Orcc XLIM and the OpenDF XLIM generation and demonstrated that actors with a lot of calculation (FDCT in our example) had almost twice the requirement in slice than the OpenDF XLIM code generation but still the Orcc XLIM throughput was 20 % higher. A possible quick fix for this problem is envisaged. Another approach which is not yet finalized is the different clock domain in actors so that a less power consumption or better performance can be achieved, depending the specification of the developer. As was mentioned in the OpenForge subsection actors can be totally asynchronous.

As for the RVC-CAL source code different metrics are being developed to help the developer to optimize its source code. A simple optimization in signal processing is the splitting of components. Here for the JPEG encoder a potential splitting of the YUV components gives a strong parallelism in the design (Fig. 9). As for coding in RVC-CAL is really easy and intuitive the programmer should take in account that its design will be data-flow by nature and potential parallelism occurs by its design.

This paper has presented a solution to generate code of the same application for hardware and software co-synthesis along with the first implementation of a JPEG Codec written purely in RVC-CAL. Orcc framework goes one step further than what was done in the OpenDF framework and it is offering software synthesis plus a better XLIM code generation. Even if a lot of work is need to be done to achieve better results both in hardware and in software level the current solution is one of the few framework in the market and in academia that can offer software and hardware code synthesis from the same specification.

6. REFERENCES

[1] XLIM: An XML Language-Independent Model. Technical report, Xilinx DSP Division, 2007.

[2] A. Antola, M. Santambrogio, M. Fracassi, P. Gotti, and C. Sandionigi. A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse c and codeveloper. In *Programmable Logic, 2007. SPL'07. 2007 3rd Southern Conference on*, pages 221–224. IEEE.

[3] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News*, 36(5):29–35, 2008.

[4] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. Overview of the MPEG Reconfigurable Video Coding Framework. *Springer journal of Signal Processing Systems. Special Issue on Reconfigurable Video Coding*, 2009.

[5] Celoxica. *Handel-C Language Reference Manual*, 2004.

[6] J. Eker and J. Janneck. CAL Language Report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, Dec. 2003.

[7] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461 – 466, jan. 2003.

[8] ISO/IEC FDIS 23001-4. *MPEG systems technologies – Part 4: Codec Configuration Representation*, 2009.

[9] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs: An mpeg-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 287 –292, oct. 2008.

[10] E. Khan, M. El-Kharashi, F. Gebali, and M. Abd-El-Barr. Applying the handel-c design flow in designing an hmac-hash unit on fpgas. *Computers and Digital Techniques, IEE Proceedings -*, 153(5):323 – 334, sept. 2006.

[11] T. J. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 224–233, New York, NY, USA, 2004. ACM.

[12] N. Siret, I. Sabry, J. Nezan, and M. Raulet. A codesign synthesis from an mpeg-4 decoder dataflow description. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 1995 –1998, 30 2010-june 2 2010.

[13] N. Siret, M. Wipliez, J.-F. Nezan, and A. Rhatay. Hardware code generation from dataflow programs. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 113 –120, 2010.

[14] C. von Platen. CAL ARM Compiler, Jan. 2010. Report of ACTORS project.

[15] M. Wipliez, G. Roquier, and J. Nezan. Software Code Generation for the RVC-CAL Language. *Journal of Signal Processing Systems*, 2009.