High Level Hardware Synthesis of RVC Dataflow Programs

Khaled Jerbi

Dedications

For the souls of my grand parents

For my Father *Abdelaziz* and my mother *Salwa* because of your invaluable love, help and encouragements during all my studies to make this PhD grade a reality, I wish you prosperity and god bless you

For my brother *Malek* who has been always there to support me and give me a smile at tough moments, I wish you all the best

For my aunt *Sabeh* because she never let me down and I know she will never do, I hope you will be always somewhere in my life

For my aunts *Monia* and *Wahida* for you supporting messages and for thinking of me as a son of you, I wish you long life and prosperity

For all my friends allover the world who cared about me and with whom I shared great moment outside my professional life

For my whole family

For everyone who cares about me

Acknowledgments

I would like to thank every person who contributed directly or indirectly to make this thesis achieved.

I would like to express my deep gratitude to all the jury members who accepted to evaluate my thesis: Mr Adel ALIMI for the honor he gave me to preside my PhD defense. I sincerely thank Mr Marco MATTAVELLI, Mr Adel GHAZEL, and Mr Jean-Philippe DIGUET for accepting to examine my research work and to participate to the to my PhD defense.

I would like to express my very great appreciation to Professor Mohamed ABID, my PhD director who offered to me this wonderful research opportunity

I would like to offer my special thanks to Professor Olivier DEFORGES, also my PhD director in France, for accepting me in his laboratory and for all these years of scientific and human exchange

I am particularly grateful for my co-director Mickaël RAULET for his exemplary supervising during the master and the PhD research and for all the techniques he taught me

I wish to acknowledge the help provided by Jocelyne TREMIER our assistant in the INSA of Rennes.

I am so thankful for Matthieu WIPLIEZ and Hervé YVIQUEL for their help on the source code of Orcc compiler.

My special thanks are extended to the staff of the CESLab and the IETR for all the great moments I had in Sfax and Rennes.

Table of contents

Part I

State Of The Art

Chapter 1

Introduction

1.1 General context

In the 50 last years, the human race has seen an amazing evolution in technology. From the black and white television till the sophisticated tablets and smart phones, the most considerable revolution rose with the emergence of embedded systems. These systems made technology very close and present in our everyday life. Nowadays, most people around the world carry on phones, smart phones, tablets and laptops. Our houses are equipped with televisions, video and music players, broadband Internet connection, alarm systems, smart washing machines etc. Our cars are fitted with GPS connection, position radar, smart weather and luminosity sensors ... This wide spread of embedded systems is firstly due to the increasing requirement of computation units and secondly to the prices proposed by the companies. In this context, the semiconductor industry association (SIA) announced that worldwide semiconductor sales in 2011 reached a record of $299.5 billion [71] with an increase of 0.4% from the $298.3 billion recorded in 2010.

All these progresses were possible thanks to the outstanding increase of the integration of transistors in the Silicon. Indeed, the number of transistors implemented on the Silicon rose from thousands to billions. For example, the number of transistors implemented on the GT200 graphic processor of Nvidia reached the 1.4 billions on a surface of 600mm^2. Even if this high level of integration made processors very powerful with high frequencies, the sequential way of execution of processors remained a continuous limitation. Therefore, designers were looking for new methods to enhance the execution performances especially with parallel architectures that can be affordable with hardware circuits. In 1987, the VHSIC Hardware Description Language (VHDL) [35] was standardized by the IEEE as a language that describes at high level parallel behaviors. This language is associated with compilers that synthesize the code into an ASIC (Application-Specific Integrated Circuit). VHDL, and also Verilog [36], are offering a powerful implementation solution to processors.

The increase of the number of transistors integrated in the Silicon allowed the developers to integrate heterogeneous architectures -processors, memories, accelerators and other hardware accelerators or IPs (intellectual Properties)- in the same chip, and the era of the System On Chip (SoC) started. These innovative architectures of the SoC allowed the developers to create complete systems on the same microscopic chip, but they rose questions about conception complexity, energy consumption, real-time performance and time to market. Consequently, designers are looking for new methods to master the complexity while keeping acceptable energy consumption and execution performances. For that purpose, the Electronic System Level design (ESLD) methodology of conception was introduced as new methodology to conceive systems in very high level called system level using virtual platforms that enable the simulation of the application at the system level. The ESLD aims to be an efficient solution to design signal processing applications at system

level and automatically generate implementations at low level.

1.2 Objectives and scientific contributions

This thesis is a joint work between the *Computers and Embedded Systems laboratory (CES Lab)* from the National School of Engineers of Sfax (ENIS Sfax, Tunisia) and the *Institute of Electronic and Telecommunications of Rennes (IETR)* from the National Institute of Applied Sciences of Rennes (INSA Rennes, France). The CES Lab main research axes are co-design, sensor nets and image processing. The IETR province in about image processing and rapid prototyping using Dataflow programming. Our research focuses on using Dataflow as a high level conception model to automatically generate hardware implementations.

A Dataflow program can be defined as a directed graph which vertices represent the execution processes and the edges represent communication FIFO channels between these processes. The data exchanged through FIFOs is called *token*. This concept makes the processes totally independent from each other and only the presence of tokens in the FIFOs is responsible of firing them. To transform the model of the Dataflow into a functional description, a domain specific language called Cal Actor Language (CAL) is considered in this work. A subset of this language is also standardized by the MPEG community as a part of the new Reconfigurable Video Coding (RVC) standard and, thus, called RVC-CAL. This standard is supported by a complete infrastructure for designing and compiling RVC-CAL programs into hardware and software implementations. However, the existing hardware generation flow presents many limitations especially for the validation and the compilation of RVC-CAL high level structures related to multi-token behavior. Indeed, for the validation, we propose a functional methodology that helps the validation of the correctness of algorithms in several steps of the conception flow. We show in this document the important impact of this methodology on the conception time.

For the hardware compilation limitations, we introduce an automatic transformation integrated in the core of an RVC-CAL compiler called Orcc (Open RVC-CAL Compiler). This transformation detects the non-compliant features for hardware compilers and makes the required changes in the intermediate representation of Orcc to obtain a synthesizable code while keeping the same global application behavior. This transformation resolved the main issue of the hardware generation from Dataflow programs.

1.3 Outline

This document is composed of three main parts:

► **The state of the art** This part contains an introduction to the global notions and research problems discussed in this thesis and the previous works that lead to our work. It contains:

- In Chapter 1 a global introduction of the motivations of this work, the objectives and the scientific contributions.
- In Chapter 2, a presentation of signal processing architectures and platforms. The methodologies to design architectures and in particular the ESLD conception methodology. The chapter ends with an overview of the existing tools for High Level Synthesis.
- In Chapter 3, an introduction to the Reconfigurable Video Coding standard and the motivation of MPEG behind this standard. It also presents the main notions and structures of the RVC-CAL language and the Model of Computation used in RVC. The last part of this Chapter is reserved for the presentation of the existing tools for the design and the compilation of RVC-CAL programs and the limitations before our contributions. It will focus especially on the explanation of the tools used during this thesis: the hardware compiler called OpenForge and the new multi-back-end compiler Orcc.

After the presentation of the state of the art and the problematic, Part II contains the main contributions and proposed techniques of this work. It contains:

- In Chapter 4, a detailed explanation of an original methodology for fast validation of RVC-CAL designs during the hardware generation. This Chapter presents also a rapid way for automatic generation of test benches and stimulus files using a software platform. This methodology is compared to the traditional flow in terms of conception duration and revealed to have an important impact on this crucial conception criterion.
- In Chapter 5, a description of the mechanisms of the automatic transformation of the high level features of RVC-CAL starting from the fundamental mathematic formalism of the actor behavior. This transformation behaves in different ways depending on the structure on the MoC of the actor, its FSM and several cases of firing rules. Therefore, all these cases are discussed and explained. To close this chapter, some miscellaneous transformations necessary for correct hardware generation and some optimizations are detailed.

The last part of the document contains the experiments and results of the application of the methodology and the automatic transformation on two application examples.

- The first technical context to test our contributions is the MPEG-4 part 2 Simple Profile video decoder. In chapter 6, the different MPEG-4 decoders available

in the RVC library are presented such as MPEG-4 part 2 SP and MPEG-4 part 10 (AVC) profiles. For MPEG-4 part2 SP, we present the serialized MVG design, the parallelized Ericsson design mainly used for hardware implementation and the Xilinx hardware oriented design. For MPEG-4 part 10, we define the different profiles (main, extended, FREXT, PHP ...) and the techniques they encapsulate. Then, an overview on the different RVC-CAL actors of these decoders is presented. Finally, the different implementation and simulation results of area consumption and time performance are summarized and compared with a reference hardware design, with an exiting IP and also with an academic hardware generation tool from C language.

- After testing a video decoder, the different methodologies are tested on a still image codec called LAR (Local Adaptive Resolution). Idem for the LAR, Chapter 7 presents the main notions and novelties of this codec and presents a set of comparison studies between the hardware generation from VHDL and the generation using RVC-CAL Dataflow programs.

Finally, Chapter 8 concludes this thesis by summarizing the scientific contributions of this thesis and discussing the perspectives and potential research activities that can be launched in the light of this work.

Chapter 2

Electronic System Level Design

In this thesis, we present a research work within the Reconfigurable Video Coding (RVC) framework defined under the MPEG-RVC standard. The purpose of this framework is to find solutions for the compilation and the implementation of Dataflow programs on embedded systems. RVC provides a domain specific language (DSL) called RVC-CAL and based on Dataflow syntax and semantics to facilitate the development of any video processing algorithm at the system level. The RVC-CAL algorithm is compiled using a compilation infrastructure into hardware and software descriptions and this methodology of conception is called Electronic System Level Design (ESLD). This Dataflow aspect substitutes the monolithic and sequential describing way of classic languages. In addition to video processing, RVC-CAL was used for still image codecs and even for network protocol IPs (Intellectual Properties) which makes it a general describing language for most signal processing contexts.

This Chapter starts with a general introduction on signal processing and more particularly on image processing. Section 2.2 presents the state of the art of the methods used by digital signal processing designers. The importance of the ESLD methodology and the related implementation techniques are illustrated in Section 2.3. The ESLD methodology enables the generation of a behavioral representation that has to be synthesized into a description of basic operations. This step is called the High Level Synthesis (HLS) and it is detailed in Section 2.4. The last Section (2.5) presents a set of tools used in the HLS.

2.1 Introduction

The digital signal processing field contains several applications such as sensor nets, radars, sonars, GPS, image etc. In the following, we will focus on the image and video processing field. This processing domain is currently omnipresent in our everyday life through television, phones, Internet ... So what is image processing and why? In 1840, William Henry Fox invented the *calotype*, a process that enables the manufacturing of light sensitive film called negative film. This film is etched by the photons (subatomic light particles) allowing the storage of the image. After that, a chemical process allows the extraction of the final photo on papers and the era of photography started. With the evolution of digital sensors in 1975, Kodak engineer Steven Sasson built the first digital camera using an array of photosensitive diodes called photosites that capture photons and converts them to electrons, much such as solar panels convert light to energy. Every diode transforms the received light into an electronic signal proportional to the light intensity and constitutes a pixel of the image. An analog to digital converter transforms these electronic signals into digital values that can be stored in memories. But what is the memory occupation of this image on the disk? Let us consider for example a matrix of sensors composed of 300x300 pixels which means 90 Kilo pixels to save. If the converter creates a digital

signal coded on 8 bits, the image size is 90 Kilo bytes. For a color image, the sensors are multiplied by three to capture the three coordinates of a color image (Red Green Blue or Red Yellow Blue) and the size is multiplied by three and reaches 270 Kilo bytes. For a video stream of one hour using the European system of 25 images/second, the size is $270 \times 25 \times 3600 = 4050000$ Kilo bytes or 4 Gigabytes! The dilemma is that the higher is the number of pixels the better is the quality of the image, but also the larger is the consumed memory. Consequently, it was necessary to find solutions to reduce the image and video size before storing them. Many algorithms have been developed to exploit the redundancy of pixels, the human eye capacities and other findings to reduce the image and video size. These techniques are called image and video compression or coding. An image may undergo other treatments for example in the medical field to extract the contours of specific anatomies or in Tele-monitoring to extract the plate number of a car etc. Therefore, we use "image processing" a more generalized word to design all these pre and post treatments.

The algorithms used for image processing are very complex and we currently speak about Intensive Signal Processing (ISP). These algorithms are executed on computing units that have three main ways to be designed: software, hardware or mixed. These architectures are explained below.

2.2 State of the art on digital signal processing conception methods

Since the second half of the 20th Century, electronic systems have progressively invaded our everyday lives after the discovery of the first transistor in 1948. Later, the evolution of the semiconductors has exploded and Dr.Gordon E. Moore, co-founder of Intel, has predicted that in 1965 when he said, in "Electronic Magazine", that the number of transistors in the same silicon area will double every two years. This evolution is presented in Figure 2.1.

It is noticeable that the most performing circuit in 1965 contained 64 transistors. When the number of transistors in the same integrated circuit is counted by thousands, we began speaking about the Large-Scale Integration (LSI) and later about the Very Large-Scale Integration (VLSI). A VLSI device may be a simple processor or a complete processing unit with a processor and all the required memories and buses. Today's integrated circuits contain billions of transistors and the Ultra Large-Scale Integration (ULSI) word is introduced.

► **Embedded systems and Systems on Chip** In accordance with Moore's law, the increasing integration rate allowed the integration of several components, representing a complete system, in only one chip. This evolution introduced the notion of

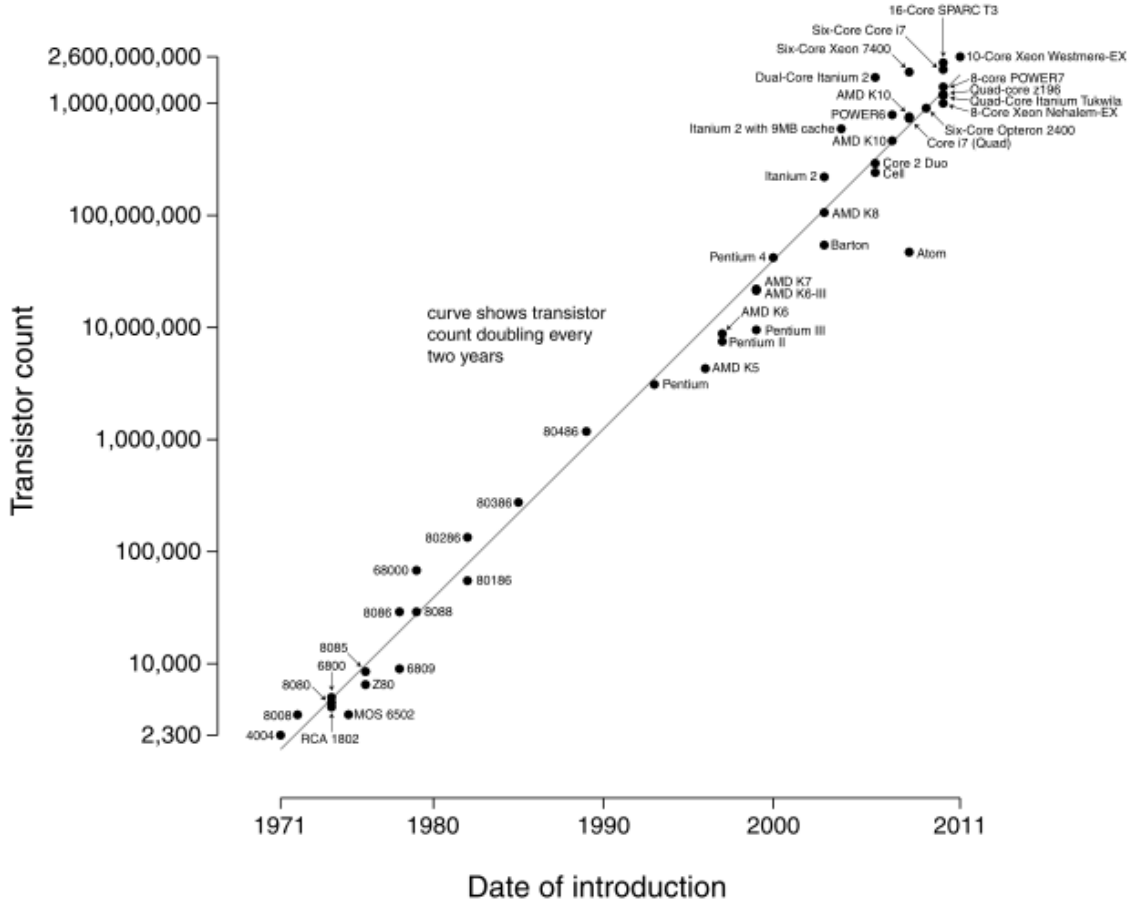

Figure 2.1: Evolution of transistors integration following Moore's law

System on Chip (SoC). The SoC technologies enable the development of application specific architectures that are optimized in terms of energy consumption, cost and performances. Consequently, such systems can be easily integrated in houses, cars or phones and the area of embedded systems has started.

On the contrary of Application Specific Integrated Circuits (ASICs), the SoC technology is able to integrate predefined blocks called (IPs) from Intellectual Properties. An IP is a circuit realizing a certain process or algorithm and that has been developed, tested and validated. The notion of collecting IPs in a SoC is very important to reduce the time to market, a criterion that revealed to be the most crucial for technology industries.

► **Parallel architectures** During many decades, the technology innovations aimed at reducing the processing time by increasing the frequency to make the system execute more statements for a given period of time. The frequency has also a

physical limit due to heat dissipation so efforts were directed in duplicating the processing units to process more data in one clock period. Others split the process to make a maximum of independent statements process in parallel. Such architectures made a revolution for ISP systems that are getting more and more sophisticated.

► **Conception methods** To master the increasing complexity of systems, many conception methods have been introduced:

- **The waterfall approach** has been elaborated in 1970 by Winston W. Royce [68]. This top-down approach considers a sequential order of development phases. To start, a set of specifications is fixed according to the needs of users. Then, an architecture is conceived and followed by implementations and assessments tests. Such organization has shown many drawbacks later. Indeed, the fact of fixing all the specifications is almost impossible because a client has generally an idea about the final product but not a detailed description of its behavior. Moreover, the complexity that appeared later made it impossible to split tasks as presented by the waterfall approach.
- **The spiral model** is defined in 1986 by Barry Boehm [9]. The principle of this approach (Figure 2.2) is to combine advantages of both top-down and bottom-up methodologies using incremental refinement.

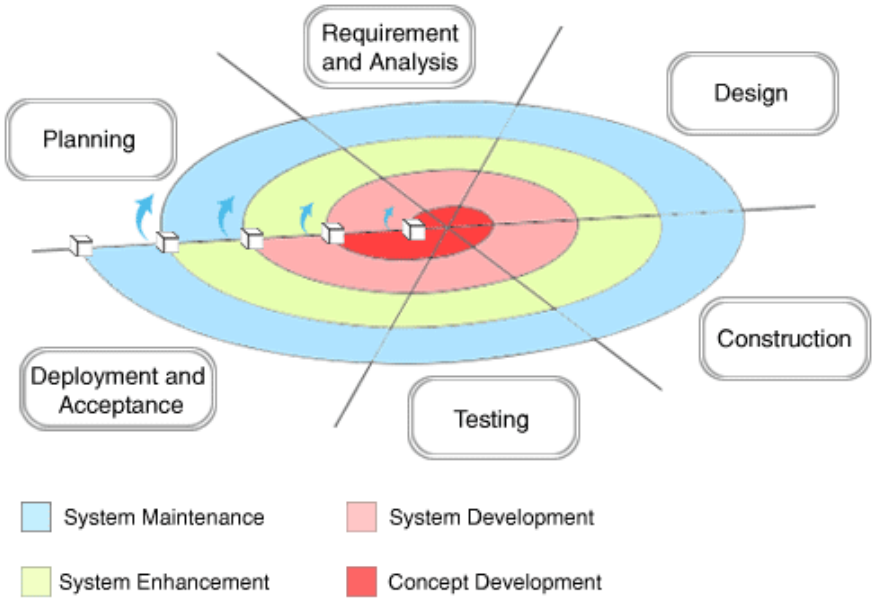

Figure 2.2: The spiral approach principal

It allows, using feedbacks when necessary (bug or system update), to realize

tests. Such flexibility of development has proved a considerable gain of time to market while keeping an excellent quality of the product. The limitation of this methodology is the fact that it is impossible to estimate the duration of the project at an advanced stage, especially for complex applications.

- **The V model** represents a sequence of steps of the project life. As presented in Figure 2.3, the main requirements are placed at the left side and their associated validation procedures at the right side. This model is a mixture of the incremental approach used for the development steps and the spiral approach used for the validation. This model is currently used in management, business, computer sciences and many other fields.

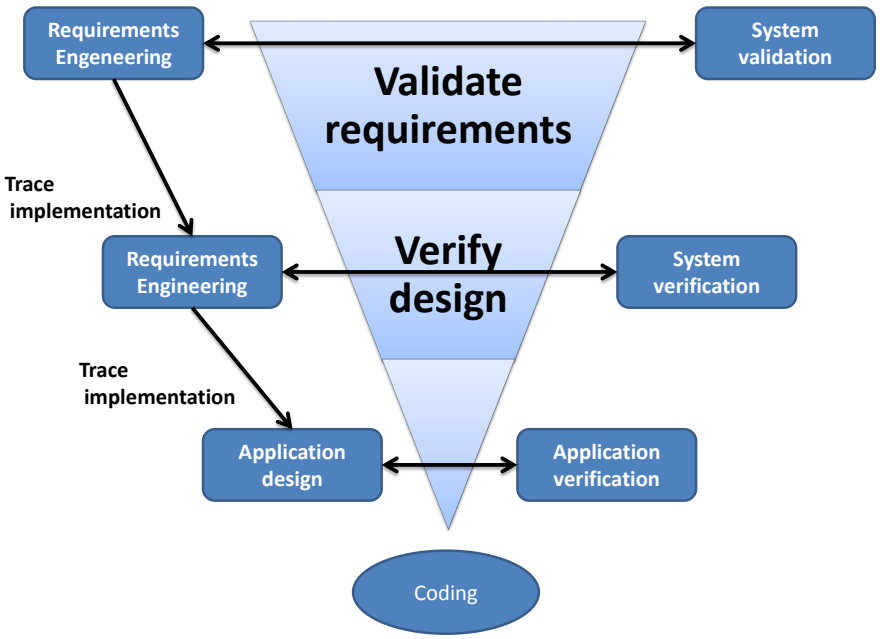

Figure 2.3: The V-Model principal

All the previously presented models continue to be used with the domination of the V-model. However, the continuously increase of applications complexity made designers think of new methods with high level of abstraction to master this complexity. In the following, we introduce the Electronic System Level Design as a methodology of conception at the system level.

2.3 Overview on the Electronic System Level Design conception method

The term ESLD methodology appeared early 21^{st} century in a semi-conductor consulting Company called Gartner Dataquest [58]. This term has been later formalized in 2007 by B.Bailey et. al in [3]. The principle is to set the conception problem of very complex applications in the highest possible abstraction level. This high level specification is then followed by High Level Synthesis (HLS) to automatically obtain a target implementation. There are two possible target architectures, software or hardware, summarized in Figure2.4 and explained below.

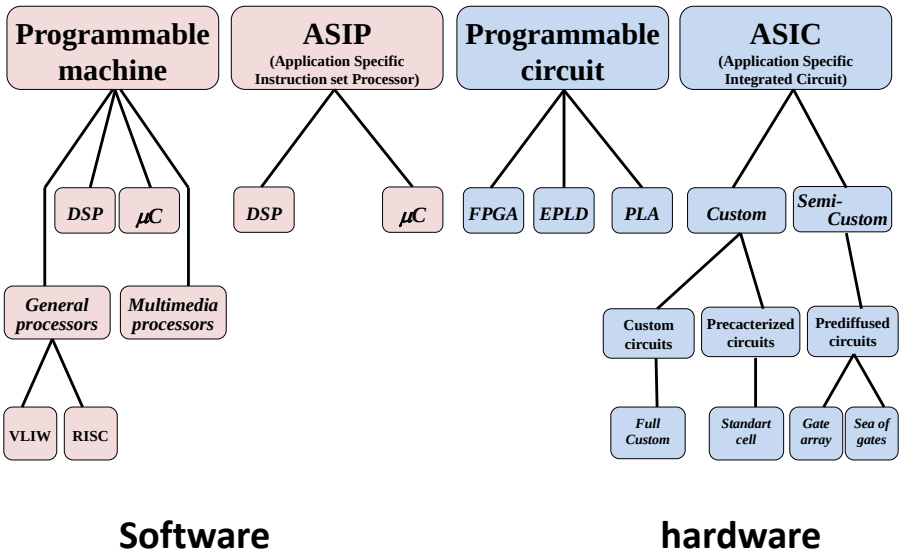

Figure 2.4: Hardware and software different targets

2.3.1 The software-oriented architecture

This architecture is based on the execution of statements on micro-processors included in architectures such as:

- general processor that we can find in any computer and it can be RISC (Reduced instruction set computing) based on simplified instructions or Very

long instruction word (VLIW) based on more complex Instruction Level Parallelism,

- DSP (Digital Signal Processor) which is a powerful processor designed for intensive computations,
- micro-controller which is a simple processor dedicated for specific simple applications,
- multimedia processor used for video display, sound treatments etc,
- the ASIP (Application Specific Instruction-set Processor) which is a full custom processor dedicated for a specific application.

The advantage of using a processor is the use of very familiar and advanced languages such as C, C++ or Java, which has a direct impact on the reduction of the development time. These languages allow also mastering very high complex applications especially the object-oriented languages. To run a program on a software target it is necessary to achieve a certain number of compilations that enable moving from the specification language to a binary code understandable by the processors. For example, the execution of a C code follows the compilation of C into assembly code then hexadecimal code and finally the binary code (see Figure 2.5).

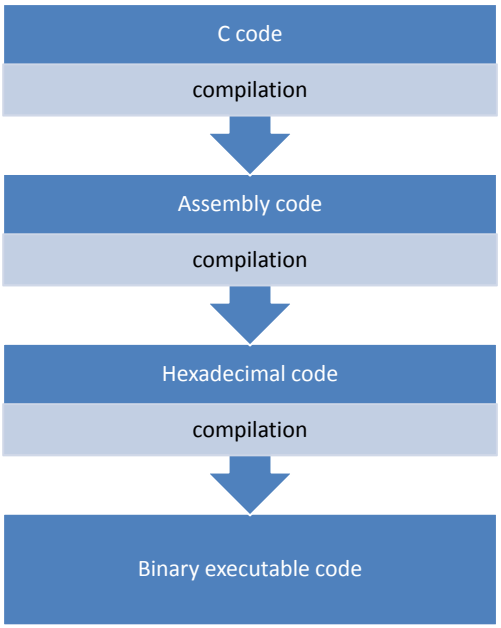

Figure 2.5: Example of C code compilation steps

Moreover, the algorithm implemented in a processor can be saved in a Read Only Memory (ROM), and almost all processors offer a way to flash this memory and put another application. This feature of software architecture is very important because it insures the flexibility and the re-usability of the target. Nevertheless, this type of circuits does have some drawbacks. They concern especially the energy consumption, the logical occupation and the performances. More precisely, a processor can never execute more than one instruction per cycle and so all statements are executed sequentially. Currently, some solutions of multi-core architectures are in progress but there is a major problem of scheduling statements into a set of processors without losing the global behavior of the application.

2.3.2 The hardware-oriented architecture

The hardware solution offers a circuit composed only of logical gates that execute basic functions directly with interconnected transistors. The revolution of hardware conception started by the appearance of languages such as VHDL and Verilog for the description of a circuit using lines of code. These languages were fist developed for military purposes but they have been later standardized and published. A hardware circuit does not encapsulate a processor. Therefore, it is possible to design parallel treatments allowing very fast architecture solutions. Moreover, many optimizations can be automatically added to reduce the area occupation. The limitations of these circuits are especially about the development time and cost, the difficulty to design very complex applications and non re-usability of the targets. These limitations are getting resolved especially with the appearance of reconfigurable circuits explained below.

The physical execution of a software program is related to the compilation and the simulation of a software language (C or Java for example) on its associated compiler. However, for hardware, things are different. The hardware implementation is finally a VHDL or a Verilog code that is going to be synthesized into a lower level code in the Register Transfer Level (RTL) that can be later transformed into transistors connections as presented in Figure 2.6 where we present the different steps of the implementation generation of an *addition* block.

The RTL level is a representation of the behavior using basic operations (addition, assign ...). This level very strategic and important because when the behavior is correct the derived circuit implementation is also correct. From this level, it is just necessary to realize the placement and Root (P&R) in the target architecture. So how to get a circuit with that code? For this purpose and as presented in Figure 2.7, hardware compilers use a library of components that allow describing any RTL (Register Transfer Level) component into a connection of logical gates and this step is called the logic synthesis. Finally, gates are automatically transformed into a connection of transistors transformed by the Silicon compiler into a mask pattern.

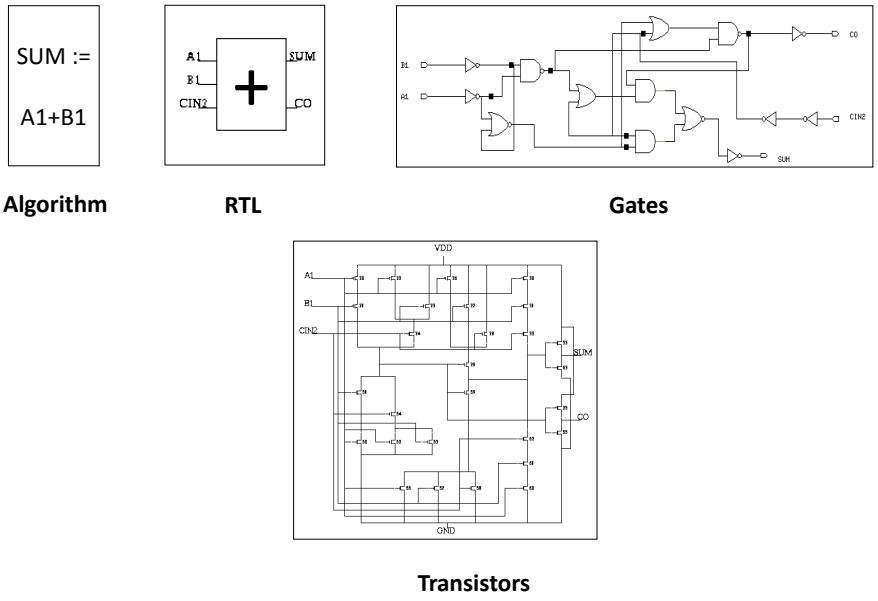

Figure 2.6: The hardware conception evolution of an addition operation block

The mask pattern is used in the Silicon foundry to create an ASIC (Application Specific Integrated Circuit) using microscopic silicon doping and etching techniques. The ASIC is the most custom hardware architecture. However, the foundry cost of ASICs creation is very high and this finding used to be the main drawback of hardware generation until the advent of the programmable circuits such as SPLD (Simple Programmable Logic Device), CPLD (Complex Programmable Logic Device) and the most recent FPGA (Field-Programmable Gate Array).

► **The SPLD** This family is one of the oldest and most basic programmable technologies. It contains architectures such as the PAL (Programmable Array logic), GAL (Generic Array Logic), PLA (Programmable Logic Array) and PLD (Programmable Logic Device). They are the smallest and the cheapest programmable logic which present simple operations such as sum and product. Their technology is similar to the PROM one. The SPLD uses fuses such as PROM to describe the logical operations which makes them non-re-programmable.

► **The CPLD** The CPLD extended the PLD concept to a higher integration level for a better system performance. It represents the equivalent of the connection of 2 to 64 SPLDs. The logical blocks communicate via programmable inter-connections presented as a matrix of programmable switches. The transistor technology usually

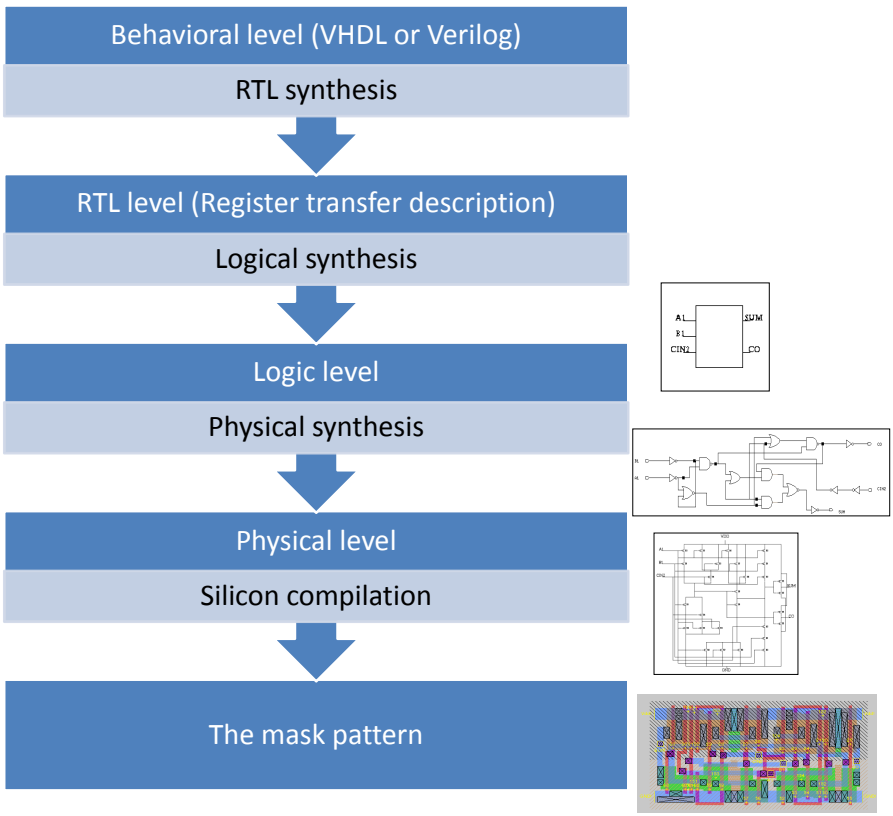

Figure 2.7: The hardware conception flow from VHDL or Verilog programs

used is the CMOS and the memory cells are EPROMs EEPROMs or Flash. The CPLD present many advantages such as:

- possible integration of complex operations,
- constant and consequently predictable propagation time,
- their macro-cells have more inputs than outputs which makes them apt for decoding operations and FSM implementations.

They also present drawbacks as:

- slow programming time,
- limited programming instructions,
- designers have to disconnect the component from the card to program the appropriate hardware.

► **The FPGA** This is the most sophisticated programmable circuits and they continue their progress thanks to the Altera and Xilinx semi-conductor companies. An FPGA is composed of logical blocks, input/output blocks and programmable inter-connections as presented in Figure 2.8. New FPGAs may contain integrated memory cells and PLL blocks (Phase-Locked Loop).

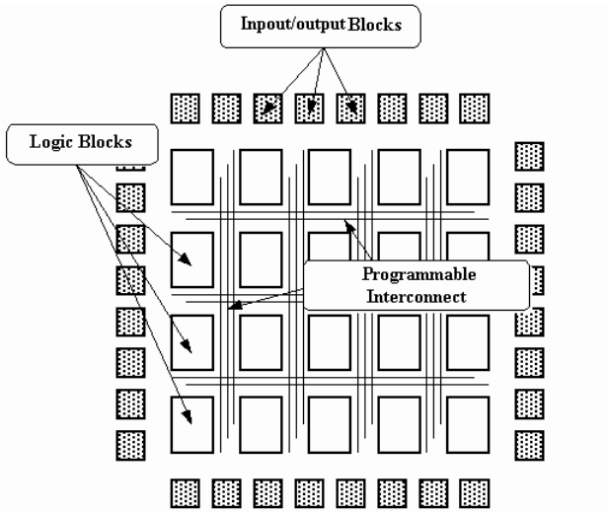

Figure 2.8: General architecture of an FPGA

The main logical blocks of an FPGA are the LUTs (Look-Up Tables) that allow the implementation of any truth table of a logical function. Every function generator behaves as a little ROM which output is selected by the input signal. The realization of a logical function with N variables requires 2^N configuration bits. As presented in Figure 2.9, a multiplexer is used later to select an output from the configuration memory.

With LUTs, we can find FPGAs with DSP blocks to perform multiplication operations.

To make the connections between the matrix of LUTs for example, the FPGAs are equipped with a memory that transforms a specific program called *Netlist* into a set of signals that control the inter-connections to make any wanted circuit. This property is very interesting since it allows the prototyping of hardware circuits before looking for Silicon foundries. The FPGA has revolutionized the world of hardware implementations and recently the software implementations. Indeed, new FPGA cards are able to integrate several embedded processors and to connect them to hardware components. For example Xilinx Company provides its tools and cards customers with processors such as Power PC or Micro Blaze. Altera Company allows users to implement software component using the NIOS processors. Researchers and industrials are now using the reconfigurable aspect of FPGAs using

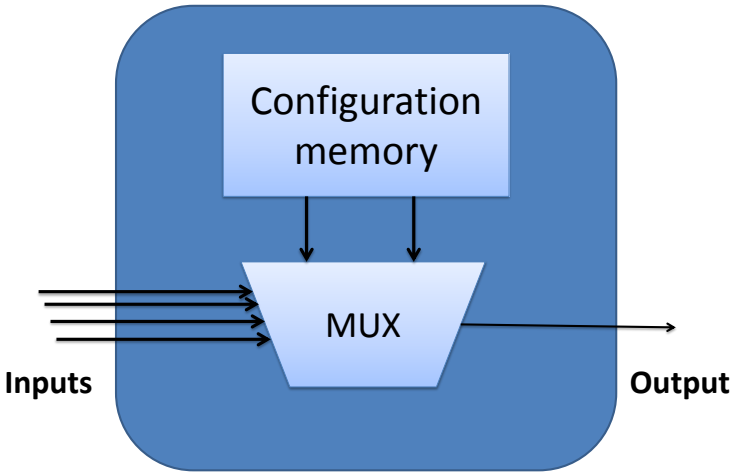

Figure 2.9: LUT architecture principle

multi-Netlist components which internal behavior change completely every configuration clock period. This reconfiguration frequency is increasing and passing over 200MHz which means 200 million circuits every second. The reconfigurable aspect is strongly related to the RVC standard, the main context of this thesis.

2.3.3 The Hardware Software Codesign

It is clear that the hardware and software architectures have both advantages and drawbacks, as summarized in Table 2.1.

	Hardware	Software
Performances	+	−
Logic occupation	+	−
Energetic efficiency	+	−
Development time and cost	−	+
complexity management	−	+
Flexibility	−	+

Table 2.1: Advantages and drawbacks of HW and SW architectures

Thus, an alternative to these two approaches have been later proposed: the Hardware Software Codesign (HSC) as an alternative trying to combine both architectures and exploit their advantages. The first attempts of HSC faced a major

problem of data communication. Indeed, software designers conceived the processors and hardware designers developed the hardware IPs separately, and they finally made the connection. Such conception flow is very time consuming due to communication and synchronization problems that involve many feedbacks. The solution was to unify the programming language and/or the conception platform [69]. For that purpose, *SystemC* was introduced as an extension of the famous C language to support hardware structures such as concurrent threads or new types (time, bit, bit_vector, logic etc.). This solution was followed by original methods such as Dataflow programming with new languages and platforms to generates mixed architectures. Test and validation platforms followed this evolution. Such well known design environments and simulators are Ptolemy [12], Vulcan [65], Cosyma [28], SynDEx, GCLP, COSYN etc. These infrastructures and platforms resolved many codesign limitations. However, the main limitation persisting is the Design Space Exploration (DSE). This codesign step consists of choosing which process is going to be implemented in hardware, and which one is going to be implemented in software. Theoretically, in the first hand, a control based algorithm that contains many statements such as *If*, *While* or *For*, should be developed in software because a *For loop* is executed sequentially and can never be parallelized. So there is no gain to use hardware for that, while a processor can execute it. On the other hand, in supercomputing, algorithms just execute very intensive computations with logical and arithmetic operators. In that case, a hardware architecture would offer very fast circuits. Nevertheless, things are not in practice as easy because the system implies more constraints. For example, if we consider a process $P1$ with intensive computations that sends data to a process $P2$ consuming that data and executing a control-based algorithm, then we speak about two strongly linked processes (Figure 2.10). The fact of implementing $P1$ in hardware and $P2$ in software has no sense because, whatever the execution rapidity of $P1$, the output of $P2$ depends only on the execution frequency of $P2$. Consequently, it could be easier to merge both processes into the same software algorithm.

This is not the only problem to mention since many other communication limitations persist. Sometimes, in the same type of architecture, there may be a conflict for example a software task may be executed by a Digital Signal Processor (DSP), a micro-controller, a multi-core platform etc. The compromise flexibility VS performance is not easy the resolve manually. Many automatic tools were developed to automatically explore the design space and try to find the best compromise, but current results are not considered enough satisfying.

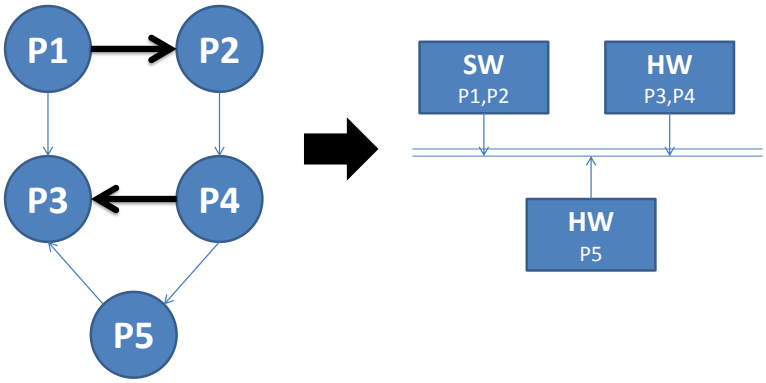

Figure 2.10: Space exploration of the design

2.4 High Level Synthesis (HLS) from high level to RTL level

The role of High Level Synthesis (HLS) tools is to transform a behavioral description (code or algorithm) into a Register Transfer Level (RTL) description. This synthesis realizes the assignment of the circuit functions to the operators so called resources, the connection between these operators and the moment each operation is executed during time intervals (called "control steps"). The synthesis can be done with two main exclusive considerations: a minimum of surface consumption or a maximum of execution speed. Let us consider a simple operation of $Y = A + B + C + D$. In the first case, the synthesis tool is going to look for the minimum of processing blocks and create the necessary scheduling to have the correct result after a set of control steps. Figure 2.11 presents one possible implementation in which one *addition* block is created.

The behavior is the following: a decoder in the scheduler fixes the multiplexer so that the first output is B and so the result of the *addition* block is $A + B$. The next control step, the MUX outputs the C and finally D. Every *addition* result is stored in the buffer so that the next operation is performed between the old result and the new value. After the necessary time, the result in Y is exactly $A + B + C + D$. At that time, the scheduler can send a signal to alert the presence of the correct

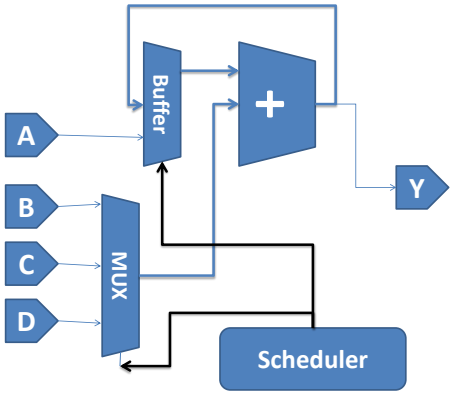

Figure 2.11: HLS with limited area constraint

result to be used for the rest of the application. It is also possible to parallelize the operations in case of data independence as presented in Figure 2.12.

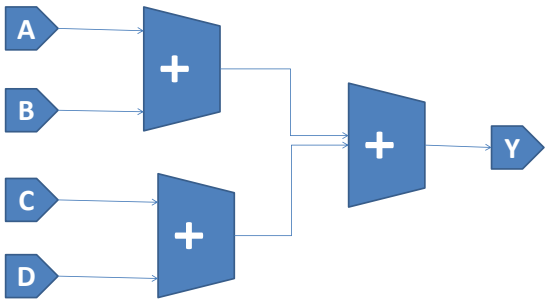

Figure 2.12: HLS with unbounded area and parallelism

The evolutions of HLS started with research tools that tried to explore a maximum of generation concepts. Later, industrials such as Mentor Graphics or Synopsys were involved but their results were not satisfying for most users because the generated code is a very low-level one, hard to debug or to optimize. After a combined effort between researchers and industrials, the third generation of tools was more successful and it is currently increasing in the market. These tools are discussed in the next section.

2.5 Existing HLS tools

Currently, the most famous HLS tools used by industrials are Catapult-C from Mentor Graphics [57], C-To-Silicon from Cadence, Symphony-C from Synopsys [75] and C2H from Altera [50], [61]. In the research field, we cite the tool GAUT [18], [17] in development in the Sticc-lab. These tools, especially the industrial ones, can generate excellent implementations from high level specification. However, the main limitation is that they cannot handle with a complete system. A partial generation applied on a simple functional block is possible but when the system gets complex, none of these tools is able to generate a correct implementation. This limitation is due to the fact that these tools take sequential codes such as C as an input. The C language has been conceived for sequential process and not hardware one. The extraction of parallelism from a code developed with a sequential philosophy can never be optimal, the generated code also. Moreover, such tools cannot accept all kinds of C code. They put many restrictions on how the C has to be written (pointers, constants, functions and main declaration). In fact, if the user intends to generate an implementation for a large application such as MPEG-AVC decoder, he is going to spend a large amount of time refactoring the code to match the specifications of the tool.

Considering this fact, new axes of research are currently oriented to the use of Domain specific Languages (DSL) that substitute C, VHDL, SystemC etc. These languages are associated with advanced design environments dedicated for codesign. In this thesis, we adopted the Cal Actor Language (CAL) and its associated compilation infrastructure for the generation of hardware and/or software implementations. This choice is explained by the fact that the CAL associated infrastructures are able to manage a complex system such as MPEG-4 Simple Profile decoder which implementation work is presented in [40]. In addition, all the tools used for graph edition, compilation and pretty printing are Open-source.

2.6 Conclusion

This chapter Introduced the Electronic System Level Design methodology. First, we presented the state of the art of the methods used for digital signal processing. These methods are not very efficient to manage the conception of a full complex application in the system level. For this reason, the ESLD is introduced as an alternative conception methodology. This methodology has to be followed by a High Level Synthesis to obtain the Register Transfer Level. Therefore, we presented the HLS techniques and a set of existing HLS tools for automatic generation of implementations from high level languages.

This work is located in the framework of the ESLD. Indeed, the main contributions of the thesis focused on finding solutions for hardware generation from

Dataflow programs. Before explaining these contributions in Part II, next Chapter introduces the Reconfigurable Video Coding standard and Dataflow compiling infrastructures.

Chapter 3

RVC: methodology and framework

In 1984, the CCITT, known currently as ITU Telecommunication Standardization Sector (ITU-T), published the first digital video decoder as H.120 based on the "COST 211" project of the Queen Mary University of London. H.120 proposed several coding methods like scalar quantization, variable length coding, differential pulse code modulation and conditional replenishment. Some advanced compression methods like motion compensation and background prediction were added in 1988. The spatial resolution results were quite satisfying but temporal performance was poor. The limitation of this codec is that it applies most algorithms using a pixel-per-pixel scan which led to the solution of block-based codecs such as H.261, considered as the pioneer of practical video coding. Later, and since the beginning of ISO/IEC/WG11 (MPEG) in 1988 with the advent of MPEG-1, many video codecs have been developed (MPEG-4 part2, MPEG-4 AVC, MPEG-4 SVC, HEVC etc.) as presented in Figure 3.1.

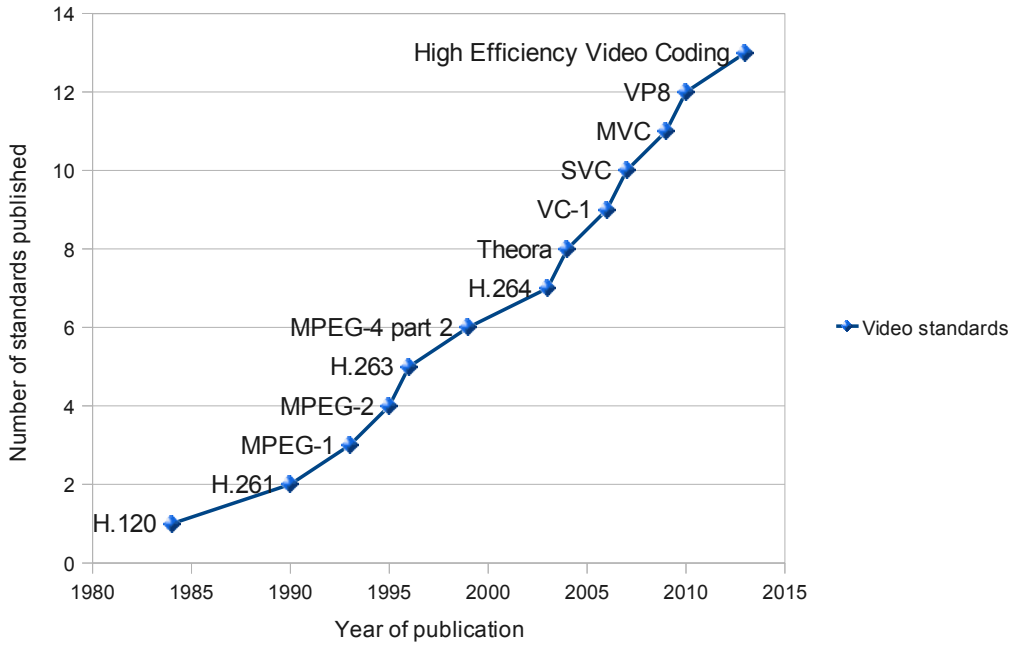

Figure 3.1: Video codecs timeline

These codecs are increasing in complexity since they proposed advanced methods for texture and motion coding and so they require larger design time. Consequently, it became a tough task for standard communities to develop, test and standardize a decoder at any given time. In addition, every standard has a set of coding techniques depending on the implementation target or the user specifications (professional or consumer) so it is not possible to implement all of them in the same

coder. That is why, standards define some algorithm subsets so called ”profiles”. A profile may encapsulate another profile or they can be completely different.

Moreover, all the previously presented standards are developed in a monolithic way making it harder to generate hardware implementations, and also to reuse or update some existing algorithms. Indeed, several standards are sharing many algorithms like a Discrete Cosine Transform (DCT) [1] or quantization but these commonalities can neither be directly exploited at the specification step, nor at the low level implementation one because of the monolithic structure of algorithms. Consequently, the whole old standards have unfortunately to be substituted by the new ones which involves also to change the player device. Such a change involves a costly replacement of users decoding multimedia devices to follow the evolution of the standards which is annoying for both users and professionals (Figure 3.2(a)). These facts originated a new conception methodology standard called Reconfigurable Video Coding (RVC) introduced by MPEG. Unlike all the other MPEG standards, RVC does not standardize algorithms to encode data but instead it standardizes the way an algorithm has to be written to conserve an independent behavior and consequently the re-usability. Thus, whatever the future video coding standard, the same multimedia support is able to decode the information as presented in Figure 3.2(b). MPEG-RVC principles and main notions are detailed in the following Section.

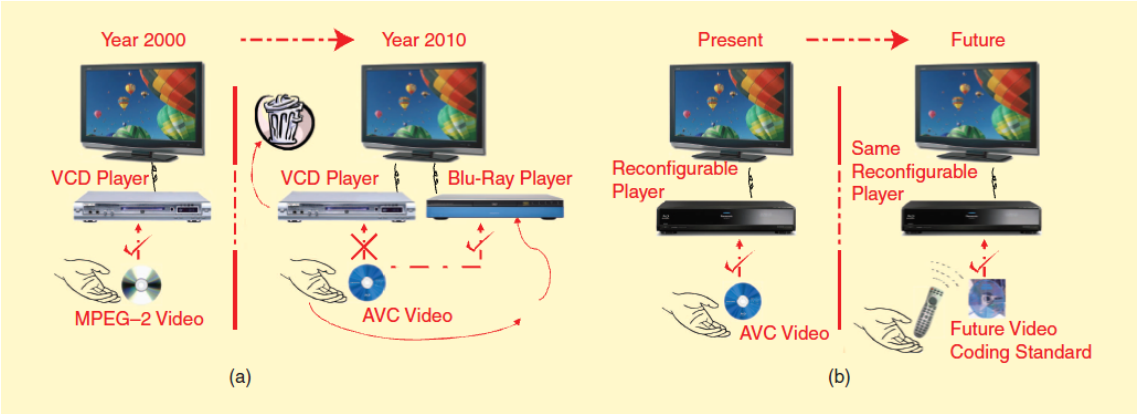

Figure 3.2: Objective of the MPEG RVC standard

3.1 MPEG RVC standard

The MPEG-RVC framework is an ISO/IEC standard under development aiming at replacing the monolithic representations of video codecs by a library of components. For that reason RVC is based on two standards:

► **ISO/IEC23001-4 or MPEG-B pt. 4** [37] which defines the framework and the standard languages to develop any RVC decoder.

► **ISO/IEC23002-4 or MPEG-C pt. 4** [38] which represents the set of employed tools.

As presented in Figure 3.3, MPEG-B is used to transform a decoder description into an abstract decoder model designed with the standardized languages (algorithms and network). It possible to use algorithms defined in the tool library of MPEG-C. This standard description represents the normative part. In the informative part, compilation solutions are used to transform the RVC design into an implementation that represents the decoding solution.

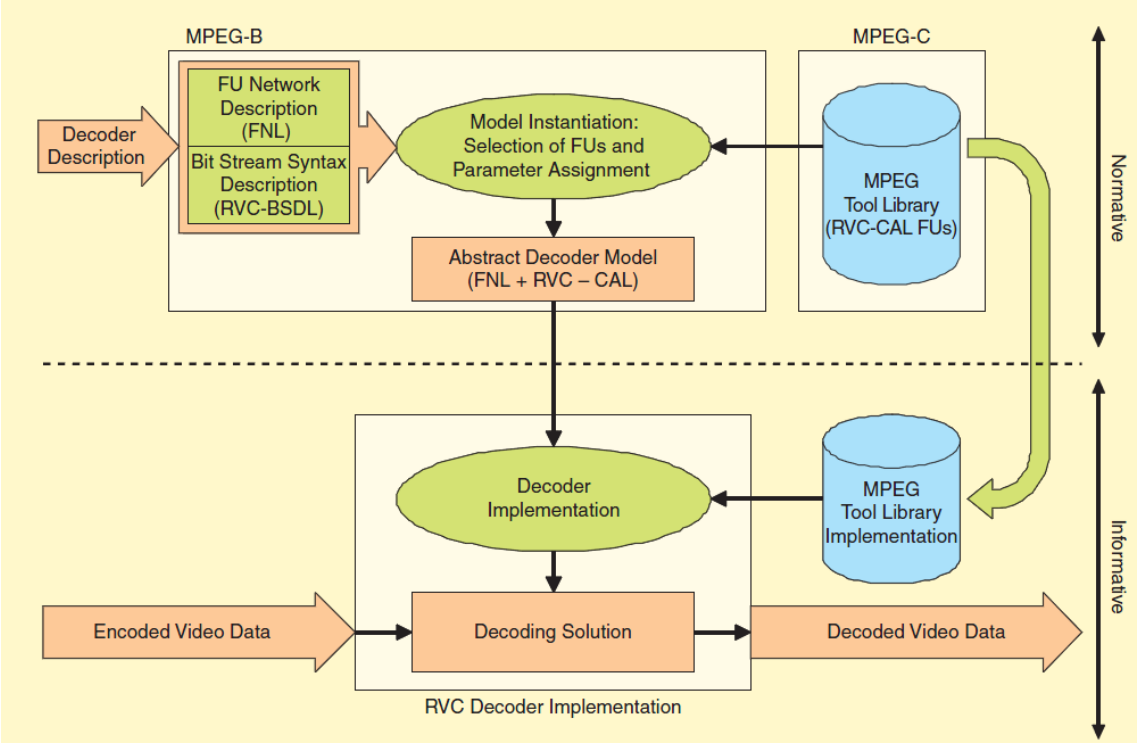

Figure 3.3: RVC framework components

RVC presents a modular library of elementary components (actors). The most important and attractive features of RVC are reconfigurability and flexibility. An RVC design is a Dataflow directed graph with actors as vertices and unidirectional FIFO channels as edges. An example of a graph is shown in Figure 3.4. This figure is an RVC description of MPEG-4 AVC decoder. The directed graph contains actors

(*demux*, *select*, *add*), FIFO channels (such as the FIFO *x* between *select* and *add*) and also directed graphs that contain actors and FIFO channels (*texture decoding*, *intra predictions* ...). Every directed graph executes an algorithm on sequences of tokens read from the input ports (*mv*, *mb_type*, *coef*) and produces sequences of tokens in the output ports (*out*).

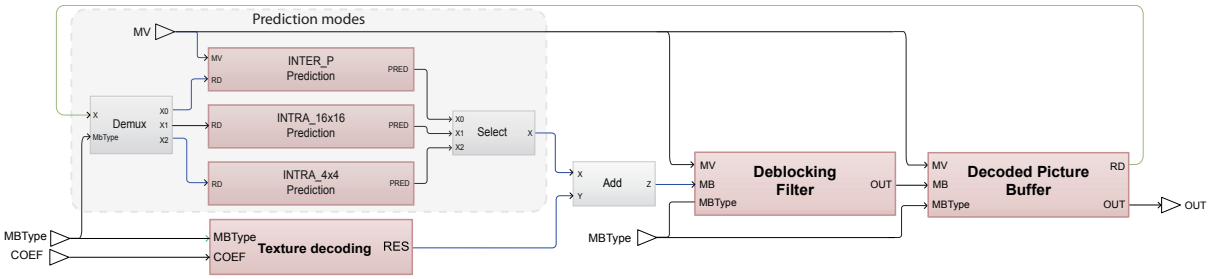

Figure 3.4: Graph example: Dataflow diagram of the MPEG-4 part 10 AVC decoder

Actually, defining several implementations of video processing algorithms using elementary components is very easy and fast with RVC since every actor is completely independent from the rest of the other actors of the network. Every actor has its own scheduler, variables and behavior. The only way of communication of an actor with the rest of the network are its input ports connected to the FIFO channels to check the presence of tokens. Then, an internal scheduler enables or not the execution of elementary functions called actions depending on their corresponding firing rules (see Section 5.3). Thus, RVC insures concurrency, modularity, reuse, scalable parallelism and encapsulation. In [40] Janneck et. al. shows that, for hardware designs, **RVC standard allows a gain of 75% of development time** for hardware design compared to existing HDLs, and also considerably reduces the number of lines of code. To manage all the presented concepts of the standard, RVC presents a framework based on the use of:

- RVC-CAL a subset of the CAL actor language called RVC-CAL that describes the behavior of the actors (detailed below in Section 3.2).

- FNL a language describing the network called FNL (Functional unit Network Language) that lists the actors, the connections and the parameters of the network. FNL is an XML dialect that allows a multi level description of actors hierarchy. It means that a functional unit can be a composition of other functional units connected in another network. For the network example of Figure 3.5, we have 3 actors *data*, *process* and *storage* with simple connections using two FIFOs. The FNL code is presented in Figure 3.6.

Figure 3.5: RVC network example

```
<XDF name="Example">
    <Instance id="Process">
        <Class name="Algo_process"/>
    </Instance>
    <Instance id="Data">
        <Class name="Algo_Data"/>
    </Instance>
    <Instance id="Storage">
        <Class name="Algo_Storage"/>
    </Instance>
    <Connection dst="Data" dst-port="o1" src="Process" src-port="i1"/>
    <Connection dst="Process" dst-port="o2" src="Storage" src-port="i2"/>
</XDF>
```

Figure 3.6: FNL code of Figure 3.5

- BDL bitstream Syntax Description Language (BSDL) [41, 55] to describe the structure of the bitstream.

- VTL an important Video Tool Library (VTL) of actors containing MPEG standards. This VTL is under development and it already contains 3 profiles of MPEG 4 decoders (MPEG-4 part 2 Simple Profile, MPEG-4 part 10 Progressive High Profile and MPEG-4 part 10 Constrained Baseline Profile).

- Tools around RVC for edition, simulation, validation and automatic generation of implementations:

- OpenDF framework [7]: is an interpreter infrastructure for the simulation of hierarchical actors network.

- OpenForge: is a hardware compiler called OpenForge[1] [34] to generate HDL implementations from RVC-CAL designs.

- Open RVC-CAL Compiler (Orcc)[2] [41]: Orcc is an RVC-CAL compiler under development. It compiles a network of actors and generates code for both hardware and software targets. Orcc is based on works on actors and actions analysis and synthesis [67, 81]. In the front-end of Orcc, RVC-CAL actors are parsed into an abstract syntax tree (AST), and then transformed into an intermediate representation (IR) that undergoes typing, semantic checks and several transformations in the middle-end and in the back-end. Finally, a code generation process is applied on the resulting IR to generate a chosen implementation language (C, Java, XLIM, LLVM etc.).

► **The importance of RVC-CAL** At this level, the question is: **why RVC-CAL and not C**? Actually, a C description involves not only the specification of the algorithms but also the way inherently parallel computations are sequenced, the way data are exchanged through inputs and outputs, and the way computations are mapped. Recovering the original intrinsic properties of the algorithms by analyzing the software program is impossible. In addition, the opportunities for restructuring transformations on imperative sequential code are very limited compared to the parallelization potential available on multi-core platforms. For these reasons, RVC adopted the CAL language for actors specification.

► **The reconfigurable aspect of RVC-CAL** The previously presented aspect of RVC is the automatic generation of hardware and software implementations using compilation infrastructures like OpenDF or Orcc. Another very important aspect of the standard is the reconfigurability. Indeed, the RVC representation of a decoder clears interoperability problems between algorithms providers and receivers. It is also possible to use proprietary Video Tool Libraries containing functional units that are not specified in MPEG-C part 4. Thus, it is possible to apply run-time dynamic reconfigurations just by modifying the topology of the network. Indeed, all MPEG decoders are based on the same hybrid decoding schemes including *intra* and *inter* predictions with more or less actors. The objective of MPEG-B part 4 is to provide a decoder for any bitstream coded with RVC specification using this specification, the hybrid decoder and actors from the VTL of MPEG-C and the proprietary VTL

[1] Available at http://openforge.sf.net
[2] Available at http://orcc.sf.net

of the intended decoder as presented in Figure 3.7. The modular representation of networks facilitates such operation.

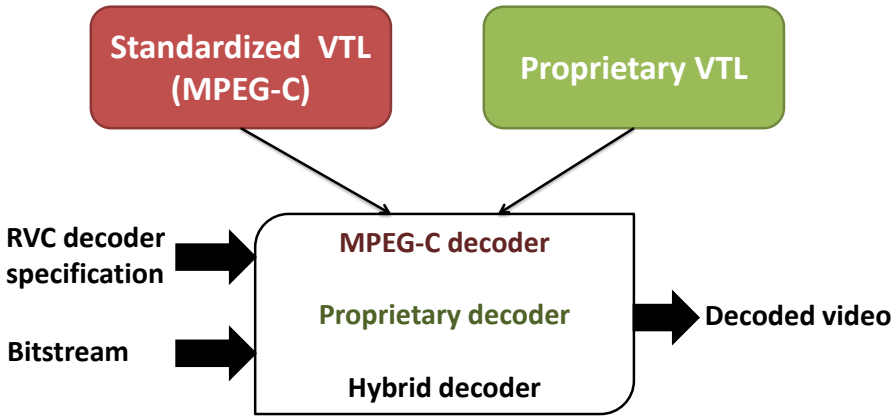

Figure 3.7: Reconfigurable principle of RVC

3.2 RVC-CAL language

RVC-CAL is a language standardized by MPEG as part of the RVC standard, and it is a restricted subset of the CAL Actor Language that was created in the *Ptolemy II Project* [11] by Janneck and Eker who detailed the rationale of the language in the CAL white paper [27] and the technical report [26]. In the following, we present the main notions of RVC-CAL:

► **Global structure of an actor** An RVC-CAL code is the description of functional unit called ”actor”. Like the notion of ”Entity” in hardware development, an actor definition begins with a header containing the specification of a box in a macroscopic way by presenting the name of the actor, parameters, ports (inputs and outputs) and their types. Figure **??** presents an actor called ”example” with a

parameter ”m” of type integer, two input ports ”IN1” and ”IN2” of type integer coded on 8 bits and one output port ”OUT” of type integer coded on 13 bits.

```
actor example (int m)
int (size=8) IN1, uint (size=12) IN2 ==> int(size=13) OUT :

  // algorithm

end
```

Figure 3.8: RVC-CAL actor header

The execution of an RVC-CAL code is based on the exchange of data tokens between actors. Each actor is independent from the others since it has its own parameters and finite state machine if needed. Actors are connected to form an application or a design, this connection is insured by FIFO channels. Executing an actor is based on *firing* elementary functions called *actions*. This action firing may change the state of the actor. An RVC-CAL Dataflow model is shown in the network of Figure A.1.

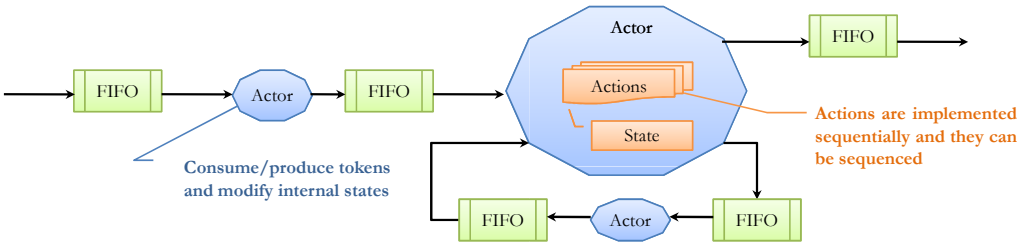

Figure 3.9: CAL actor model

Figure **??** presents an example of a CAL actor realizing the sum between two tokens read from its two input ports.

In the “sum” actor, the internal scheduler allows action “add” only when there is at least one token in the FIFO of port “INPUT1” and one token in the FIFO of port “INPUT2”. This unique dependency from the presence of data in the FIFOs explains the fact that an actor can neither read nor modify the state of any other actor.

▶ **Variables** Before starting the micro-definition of an actor process, it is possible to declare state variables that may be shared by all the actions of this actor.

```
1  actor sum ()
2  (int size=8) INPUT1, (int size=8) INPUT2 ==> int(size=8) OUTPUT:
3
4   add: action INPUT1:[ i1 ], INPUT2:[i2] ==> OUTPUT:[s]
5   var
6     int s
7   do
8     s:= i1 + i2 ;
9   end
10 end
```

Figure 3.10: Example of sum actor

Of course, actions may have their own local variables. As presented in Figure 3.11, variables may be integers (line 1), unsigned (line 2), boolean (line 3) or lists (line 4). A list in CAL is an array of elements of the same type (int, uint or bool). As it is intended for both hardware or/and software implementations, it is necessary to specify the exact dynamic of variables and ports (line 5).

```
1  int var1 := 0;
2  uint var2;
3  bool var3 := true;
4  List (type:int, size = 10) var4;
5  int (size=12) var5;
```

Figure 3.11: Example of variables declaration

► **Expressions** Like any programming language, RVC-CAL uses a set of expressions that can be mathematical (Figure 3.12.line 1) or binary (Figure 3.12.line 2). The only restriction is that these expressions are side-effect free which means that they are idempotent, and consequently they cannot change the value of external elements like state variables.

```
1  x:= a + 5;
2  y:= a & 7;
```

Figure 3.12: Examples of expressions in RVC-CAL

Expressions are used to form the statements of an algorithm. RVC-CAL presents five types of statements:

- The assignment: an expression may be assigned to a variable (local or global). It is also possible to assign it to a list index in case of lists.

- The call: this statement is used when functions or procedures are used in the algorithm. The call of a function is accompanied with an assignment of the result of this function to a state or local variable.
- The While loop: for an unknown or infinite number of executions. The loop continues the execution while a conditional expression is true.
- The foreach loop: for a finite number of executions.
- The ”if .. then .. else”: for a conditional execution of the statements.

► **RVC-CAL action** An action represents the microscopic representation of the behavior. Every action is related to a *firing rule* which is the rule that specifies the necessary conditions that allows the action to be executed. These conditions represent the *schedulability* and they concern especially the number of available tokens in the input FIFOs. Other conditions on the value of input tokens or the value of a state variable may be added using the ”guard” instruction. The importance of this additional condition is that it is tested before consuming data from the FIFOs. As shown in Figure 3.13, the firing rule is presented in the header of the action followed by the body. The header may also contain the declaration of local variables.

Scheduling condition

```
1 action_name: action IN1:[in1], IN2:[in21, in22] ==> OUT:[o]
2 var
3   int o := 0
4 guard
5   in1 > 0,
6   in21 = counter
```

Body

```
1 do
2   o := counter +1;
3 end
```

Figure 3.13: Action main parts: scheduling condition and body

To be executed, the action of Figure 3.13 has to satisfy the scheduling condition of: presence of one positive token at least in the FIFO of Port ”IN1”, two tokens in the FIFO of port ”IN2” such as the first of them equals the value of a state variable ”counter”. When the scheduling conditions are true, the action consumes the 3 tokens from the FIFOs, executes the body and outputs the value of ”o” in the FIFO of port ”OUT”. Now, if we substitute the guard condition by a conditional block, the algorithm becomes:

```
1 action_name: action IN1:[in1], IN2:[in21, in22] ==> OUT:[o]
2 do
3   if in1 > 0 && in21 = counter
4         then
```

```
5       o := counter + 1;
6     end
7   end
```

In this case, the presence of tokens is enough to fire the action which means that the three tokens are already consumed before the ”if” test. If the ”if” condition is false, the action is not going to execute any statement and the output in the port ”OUT” will be the old value of the variable ”o”. The consumed tokens cannot be used anymore by any another action.

▶ **Functions** The body of an action or even the ”guard” condition may call a set of expressions located in a function. The function is considered as an expression which type is the type of its return. Consequently, it is also side-effect free. A function header (line 1 of Figure 3.14) presents the tag, the parameters and the type of the result; then follows the body containing the algorithm.

```
1  function divroundnearest(int i, int iDenom) --> int :
2    if (i >= 0) then
3      (i + (iDenom >> 1)) / iDenom
4    else
5      (i - (iDenom >> 1)) / iDenom
6    end
7  end
```

Figure 3.14: Example of a function in RVC-CAL

▶ **Procedures** Concerning the procedure and like many other imperative languages, it represents a set of side-effect instructions that can be called at any time by an action or a function or another procedure. The RVC-CAL code of a procedure has the form shown in Figure 3.15.

```
1  procedure procedure_tag(parameters)
2  begin
3    // instructions
4  end
```

Figure 3.15: Example of a procedure in RVC-CAL

► **Actions priority** The behavior of an actor is managed by a global action scheduler that detects the schedulability of all actions and allows the firing of the action which firing rule is true. Nevertheless, it is possible to have two or more schedulable actions at the same time and the question is which action will be allowed by the global scheduler? If the choice is random, we will have a non-deterministic behavior of the actor. For this purpose, the notion of priority has been added to RVC-CAL using the following structure of Figure 3.16:

```
priority
  action1 > action2;
  action3 > action4 > action2;
end
```

Figure 3.16: Example of a priority in RVC-CAL

More details are presented in section 5.3.

► **Finite state machine (FSM)** For more complex actors, RVC-CAL offers the possibility to use an FSM scheduler to add more restrictions on the actions that can be scheduled at a given state. An FSM has an initial state and every state presents a set of actions that may let the actor in the same state or that may change the actor state. In case of conflicted actions in the same state, it is necessary to add a priority. Figure 3.17 shows an RVC-CAL declaration example of an FSM that can be represented graphically by the schema of Figure A.3 where states are presented with vertices and the transitions with edges. This type of representation will be adopted for the rest FSM related figures.

```
schedule fsm init_state:
  init_state     ( action1 ) --> state10;
  state10        ( action2 ) --> state10;
  state10        ( action3 ) --> state11;
  state11        ( action4 ) --> init_state;
end

priority
  action3>action2;
end
```

Figure 3.17: Example of an FSM in RVC-CAL

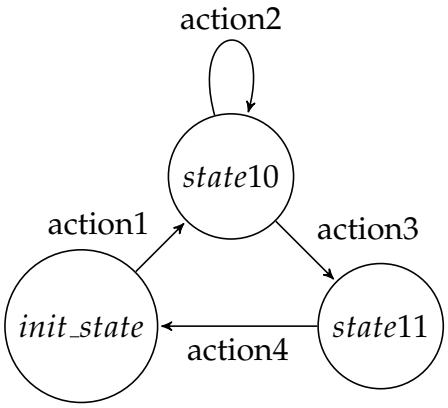

Figure 3.18: FSM graph representation

► **The untagged actions** An action may be included in a finite state machine or outside the FSM when it has no label. An action in the FSM is fired when its schedulability is true and when it is in the list of the executable actions of the current state of the FSM. Whereas, the action outside the FSM, so called *untagged*, fires when the schedulability is true whatever the current state of the FSM. The untagged action is not called in the FSM but it is considered with higher priority by the actor scheduler. In the example of Figure 3.19, the actor A presents an untagged action. The FSM starts at the state $S0$. If there is no token in the port I then the *process* action fires and the FSM state is updated to $S1$. At this state, the action *write* writes the 64 tokens of the table *tab* in the port O, but, if at any moment, a sequence of tokens is present in the port I then the *write* action is not executed and the untagged action fires.

Generally, this kind of actions is used when the execution of an action is crucial whatever the state of the FSM. The equivalent of an untagged action a is to create for each state s of the FSM a transition $s(a) - - > s$; and to set a with a higher priority than the rest of the actions. These properties of the untagged action revealed to be very important in the contribution of Chapter 5.

► **Software and hardware RVC-CAL-oriented implementation** For the same behavior, an actor may be defined in different ways. Let us consider the "sum-5" actor of Figure **??** that reads 5 tokens in a port "IN", computes their sum and produces the result in a port "OUT".

In Figure **??**(a), the required algorithm is defined in only one action. The condition of 5 required tokens is expressed by the instruction "repeat 5". Action "add" fires by consuming the 5 tokens from the FIFO into an internal buffer "i". After data storage, the algorithm of the action is applied. Finally the action firing finishes by

```
actor A () int I, bool S ==> int O :

  bool s := false ;
  int data := 0;
  List (type: int, size = 64) tab;
  int counter := 0;

  action I:[input] ==>
  guard counter < 64
  do
    tab[counter] := input ;
    counter := counter + 1;
  end

  process: action ==>
  do
    f(tab); \\ f a considered function
  end

  write: action O:[tab] repeat 64
  do
    counter := 0;
  end

  schedule FSM S0:
    S0(process)-->S1;
    S1(write)-->S0;
  end

end
```

Figure 3.19: RVC-CAL example of an actor with an untagged action

writing the result in the port "OUT". Such description is very fast to develop and implement on software targets but for hardware implementations a multi-token read is not appropriate. This is the reason of developing the equivalent mono-token code of Figure **??**(b). In this description, we use a finite state machine to lock the actor in the state "state0". While counter < 5, only the action "read" can be fired to store tokens one per one in "data" buffer. Once the condition of action "read_done" (counter = 5) is true, both of "read" and "read_done" actions are fireable. This is why the priority "read_done $\succ$ read" is important to keep the determinism of the actor. Finally, the firing of "read_done" action involves an FSM update to "state1" where only "process" action can be fired and the actor is back to the initial state.

3.3 RVC Models of Computation

The Dataflow model represents the main line of research of this thesis. As a definition, a Dataflow program is a conception method for signal processing units. Unlike the imperative programming that considers that the exchanged data between operations is secondary to the behavior of the operations themselves, the Dataflow

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 add: action IN:[ i ] repeat 5
 ==> OUT:[ s ]
 var
   int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + i[k] ;
   end
 end
end
```

(a) SW oriented definition

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 List (type: int (size=8), size = 5) data;
 int counter :=0 ;

 read: action IN:[ i ] ==>
 do
   data[counter] := i ;
   counter := counter + 1 ;
 end

 read_done: action ==>
 guard
   counter = 5
 do
   counter := 0 ;
 end

 process: action ==> OUT:[ s ]
 var
  int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + data[k] ;
   end
 end

 schedule fsm state0:
   state0 (read)      --> state0;
   state0 (read_done) --> state1;
   state1 (process)   --> state0;
 end

 priority
   read_done > read;
 end

end
```

(b) HW oriented definition

Figure 3.20: Two-way definition example of sum-5 actor behavior

presents a model that puts independent operations (processes) in the first concern and the connections in a secondary importance. This type of designs is very important for parallel programming. The first who introduced this model is Bert Sutherland in his Thesis in 1966 [74] with the idea that the fact of changing the value of a variable during the process involves the update of all variables values related to that changed variable. Later many research laboratories were interested like *Supercomputer Labs* and *Lawrence Livermore National Laboratory* where the most popular Dataflow language (at that time) called *SISAL* [49] is developed. To make the programming more popular *SAC* (Single Assignment C) language [63] is developed to design Dataflow as close a possible to the C language. In early 1990s, a revolutionary tool called *Prograph* [20] is introduced as a visual, object-oriented, Dataflow, multi-

paradigm programming language that uses iconic symbols to represent actions to be taken on data. Late 1990s, rose the *National Instruments LabVIEW* language which was intended to connect laboratory equipments but finally it was generalized for all signal processing. Another tool designed for digital sensors and equipments called *VEE* [33] was introduced in 1991 and continues progressing until the 9.3 version of Nov 2011.

In the beginning of the 21^{st} century, an important research project in the university of Berkeley, introduced the Cal Actor Language and proposed a set of tools to automatically generate software and hardware architectures. This infrastructure has an important impact on this thesis and will be detailed later in this chapter.

3.3.1 Overview

A Model of Computation (MoC) is defined in the literature of the computability theory as the set of operations allowed to describe an algorithm. MoCs can be related to Turing machines, lambda calculus or Dataflow. In this thesis, we consider only the Dataflow MoC since it meets the needs of the RVC standard. The Dataflow graph is a directed graph defined by a couple $G = (V, E)$ where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. An edge is a couple $e = (i, j)$ such as i is the initial extremity of e and j in the final extremity of e. We define $\mathsf{src}(e) = i$ and $\mathsf{dst}(e) = j$. The set of predecessors of a vertex j is defined with $\mathsf{pred}(j) = \{i \in V \mid (i, j) \in E\}$. Mutually, the set of successors of a vertex i is defined with $\mathsf{succ}(i) = \{j \in V \mid (i, j) \in E\}$. A Dataflow graph of an operation " (a+b)x(a-b) " is presented in Figure 3.21.

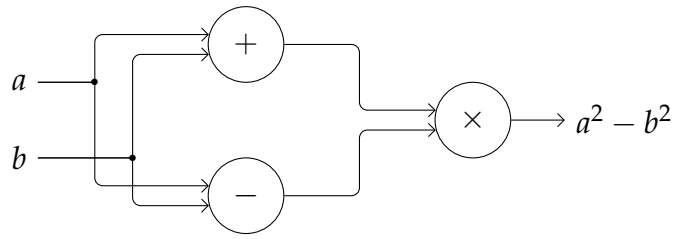

Figure 3.21: Dataflow graph of operation $y = (a + b) \times (a - b)$

RVC standard is based on a Dataflow MoC called Dataflow Process Network (DPN) [53] related to the Kahn Process Network (KPN) [48]. In [48], Kahn introduced the KPN as a distributed Dataflow MoC which is a common model for describing signal processing systems and modeling distributed systems and parallel programming. In this model, the vertices are a set of deterministic processes and the edges are unbounded FIFO channels (Figure 3.22).

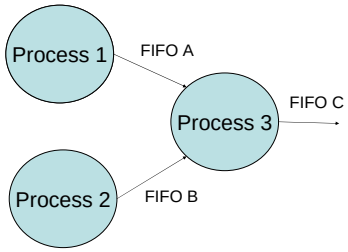

Figure 3.22: KPN example of three processes and three FIFOs

A KPN is a continuous and deterministic network. The execution of the KPN model is generally divided into a set of threads. The execution of any thread consists of three main parts: a blocking read of the required data of a process in the input FIFOs, an execution of the process and a write of the results in the output FIFOs. All these steps can be interrupted by other actors at any time. Figure 3.23 presents an FSM representation of the behavior of a thread executed in a KPN. The thread starts at the state *wait for data* to read the necessary number of tokens from the input FIFOs. When the required number is obtained, it is possible to move to the state *execution*, realize the process computations and then write the results in the output ports and go back to the state *wait for data* to read new tokens.

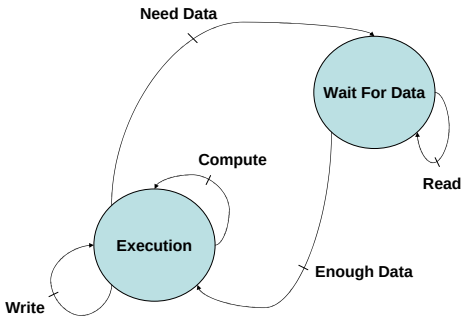

Figure 3.23: The FSM representation of every process behavior in a KPN model

To understand the limitation of this MoC, Figure 3.24 shows the Petri Net representation of the behavior of *process*3 already presented in Figure 3.22. The execution starts by reading from $FIFOA$. When the required number of tokens from that FIFO is obtained, the network moves to reading from $FIFOB$. After that, the process is

able to be executed as explained above. It is obvious in the Petri net representation that KPN threads assume an unlimited stream of data to avoid being deadlocked because if there is no more tokens in ”FIFO A” or in ”FIFO B” then the state ”execution” is never reached. As the execution is based on multiple threads, the global execution of the design continues by firing other threads. But, the threads switch involves a very costly context switch and so it has to be avoided. So how is it possible to ensure that the read is no more “blocking”?

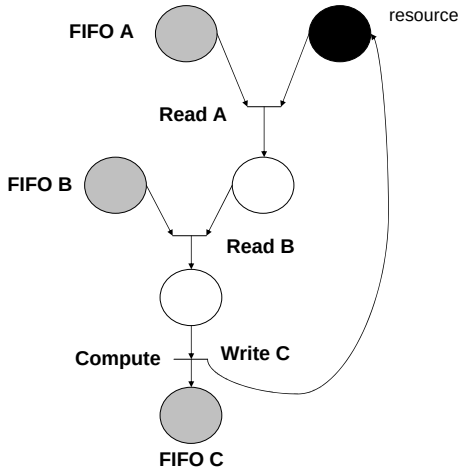

Figure 3.24: The Petri net representation of KPN mechanism

In the Following, we present an answer to this question by introducing the formalism of the DPN as a general MoC of RVC and we explain the principle of some derived MoCs with more restrictions than the DPN.

3.3.2 The Dataflow Process Network MoC and derived MoCs

The network deadlock problematic was resolved when Dennis [53] and Lee [52] extended the KPN model by introducing the concept of firing rules. The process - henceforth called actor- may be triggered by several combinations of data sequences from the input ports. Each actor triggering, so-called firing, is related to a condition of presence of a data sequence and this condition is called a firing rule. These notions originated the Dataflow Process Network (DPN). The DPN model allows actors to test an input port for absence or presence of data. Consequently, when no more data is available the KPN must wait because it is blocked in a reading state while the DPN spies on the FIFO and reads data only if enough data is present. The DPN represents the most general Dataflow MoC. Many restrictions on the dependencies may be added to the model for an easier analysis which involved other Dataflow

MoCs: the quasi-static Dataflow (QSDF), the cyclo-static Dataflow (CSDF) and the most restricted one the Synchronous Dataflow (SDF). The classification of the RVC-CAL actors is based on state dependence and data dependence. A state dependent actor is an actor which behavior is controlled by a Finite State Machine. For this type of actors, every state presents a set of actions that realize transitions from that state to another one or to the same state. A state dependent actor fires only when a transition action of the current state of the FSM is fireable. Concerning the data dependent actors, the firing rules control the value of the tokens and the fireable action is selected depending on that value. As shown in Figure 3.25, the most expressive MoC is the DPN. By adding restrictions of data or state dependence, the expressiveness decreases but, inversely, the analyzability increases.

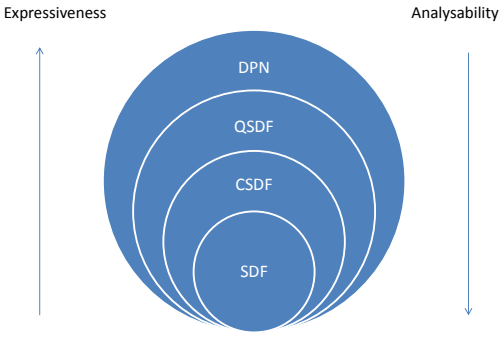

Figure 3.25: Dataflow MoCs

The classification of MoCs can be summarized in Table 3.1.

	Data dependent	Data independent
State dependent	DPN	CSDF
State independent	QSDF	SDF

Table 3.1: Actors classification in MoCs

The important property of the RVC-CAL language is the ability to manage all these MoCs.

3.3.2.1 RVC modeling of the DPN

Let Ω be the universe of all tokens values exchanged by the actors and $\mathbb{S}=\Omega^*$ the set of all finite sequences in Ω. We denote the length of a sequence $s \in \mathbb{S}^k$ by $|s|$ and the empty sequence by λ. Considering an actor with m inputs and n outputs, $\mathbb{S}^m$ and $\mathbb{S}^n$ are the set of m-tuples and n-tuples consumed and produced. For example, s_0=[λ,[t_0,t_1,t_2]] and s_1=[[t_0],[t_1]] are sequences of tokens that belong to $\mathbb{S}^2$ and we have $|s_0|$=[0,3] and $|s_1|$=[1,1]. The only information that fires an actor is the presence of enough data to satisfy one of its firing rules. Once a rule is satisfied, a corresponding local function called *action* is executed by consuming tokens from the input FIFO and producing others on the output FIFO. This action firing may change the state of the actor.

A Dataflow actor is defined with a pair $< f, R >$ such as:

* $f : \mathbb{S}^m \rightarrow \mathbb{S}^n$ is the firing function
* $R \subset \mathbb{S}^m$ are the firing rules
* For all $r \in R$, $f(r)$ is finite

An actor may have N firing rules which are finite sequences of m patterns, one per input port. A pattern is an acceptable sequence of tokens for an input port. It defines the nature and the number of tokens necessary for the execution of at least one action. RVC-CAL also introduces the notion of *guard* as additional conditions on token values. An example of firing rule r_j in $\mathbb{S}^2$ is:

$$\begin{cases} g_{j,k} : [x] | x < 0 \\ r_j = [t_0 \in g_{j,k}, [t_1, t_2]] \end{cases} \qquad (1)$$

which means that if there is a negative token in the FIFO of the first input port and 2 tokens in the FIFO of the second input port then the firing rule is satisfied and therefore an action is fireable. An action is fireable or schedulable iff :

- The execution is possible in the current state of the FSM (if an FSM exists)
- There are enough tokens in the input FIFO
- A guard condition returns true

It is very important to notice that there are two types of the dynamic actors: the time-dependent and the time-independent. The time-independent case is deterministic which means that, for some sequences of tokens in the input ports, it is possible to predict the behavior and consequently the outputs of the actor what ever the order or the instant token arrive to the port. However, for the time-dependent case, the instant tokens arrive to the FIFO can change the behavior of the actor which involves a non-deterministic behavior. In the example of the actor of Figure 3.26,

the firing rule of the action "read_signed" is the presence of a token in the FIFO of port "S" and the firing rule of action "read_data" is the presence a token in the FIFO of port "I". But, it is notable that the body of the second action uses the last saved value of the first action which means that, for the same state and at the same time, the actor can behave in a different way depending on what happens first: the arrival of a token in "S" or in "I".

```
1  actor time_dependent () int(size=10) I, bool S ==> int(size=9) O :
2
3    bool s := false ;
4
5    read_signed : action S:[ signed ] ==>
6    do
7      s := signed ;
8    end
9
10   read_data: action I:[i] ==> O:[ f(i,s) ]
11   end
12
13   priority
14     read_signed > read_data ;
15   end
16
17 end
```

Figure 3.26: RVC-CAL example of time-dependent actor

The example of figure 3.26 illustrates also a case that can be modeled by the DPN MoC and not by the KPN one. Indeed, in the KPN case, we have a reading state that expects to find the necessary tokens in "S" and "I" in a first step before the execution of the adequate action and the write of token in the output FIFOs. This execution order may deadlock the thread associated to this actor if it does not receive any more tokens in one of its input ports.

3.3.2.2 RVC modeling of the SDF

If an actor has a systematic behavior without an FSM and the nature of the data does not change the number of tokens consumed or produced the it is considered as an SDF actor [51]. In other terms, an actor is classified SDF if it has only one firing rule or if all firing rules consume and produce the same amount of data:

If $\mathbb{R}$ is the set of all firing rules then $\forall r_i, r_j \in \mathbb{R} : |r_i| = |r_j|$

SDF is obviously the easiest MoC to be analyzed. An SDF graph can be mapped on multi-core architectures because the schedulability and the memory consumption can be computed at the compilation step using bounded FIFO memories. The actor of Figure 3.27 presents an example of SDF MoC in RVC-CAL. Despite the presence

of two different actions (”rule1” and ”rule2”), the firing rules of both actions use the same input and output patterns which means that it is possible to use only one action with an ”‘if” statement that replaces the ”guard”.

```
actor sdf ()
int (size=8) INPUT1, int (size=8) INPUT2 ==> int(size=8) OUTPUT:

 rule1: action INPUT1:[ i1 ] repeat 5 , INPUT2:[i2] repeat 10  ==> OUTPUT:[o]
 var
   int o
 guard
  i1>0
 do
   o := f1(i1,i2); \\ f1 and f2 considered functions
 end

 rule2: action INPUT1:[ i1 ] repeat 5 , INPUT2:[i2] repeat 10  ==> OUTPUT:[o]
 var
   int o
 guard
  i1<=0
 do
   o := f2(i1,i2);
 end

end
```

Figure 3.27: Example of modeling SDF MoC in RVC-CAL

3.3.2.3 RVC modeling of the CSDF

For the CSDF[8], the process is composed of a set of actions that have completely different behavior, but, the global macro-behavior is static. The CSDF MoC extends the SDF by adding an FSM that loops in a static sequence of states. Therefore, the memory consumption and the scheduling information remain the same for the compile-time properties. In the example of Figure 3.28, the actions ”firstAction” and ”secondAction” are completely different in the schedulability and in the body. Nevertheless, the FSM makes always that the actor executes ”firstAction” and then ”secondAction” which is a global cyclo-static behavior.

3.3.2.4 RVC modeling of the QSDF

A quasi-static (QSDF) actor [5, 13, 6, 10, 16] presents a mutually exclusive subsets of static processes. Depending on the value of the input data the actor is going to execute one of these subsets as presented in Figure 3.29 where the data in port

```
actor csdf ()
int (size=8) INPUT1, int (size=8) INPUT2 ==> int (size=8) OUTPUT:

 firstAction: action INPUT1:[ i1 ] repeat 5 , INPUT2:[i2] repeat 10  ==> OUTPUT:[o]
 var
   int o
 do
   o := f1(i1,i2);
 end

 secondAction: action INPUT1:[ i1 ] repeat 30 ==> OUTPUT:[o]
 var
   int o
 do
   o := f2(i1);
 end

 schedule fsm s0:
   s0 (firstAction ) --> s1;
   s1 (secondAction) --> s0;
 end

end
```

Figure 3.28: Example of modeling CSDF MoC in RVC-CAL

”C” is going to determine the process to execute among processes (A,B and C) and these processes consume data from ports ”I1” and ”I2” and produce in port ”O”. If the type of the conditional data is a Boolean, we speak about a subset of the QSDF called Boolean Dataflow (BDF) MoC analogical to a multiplexer (MUX) in the electronic field.

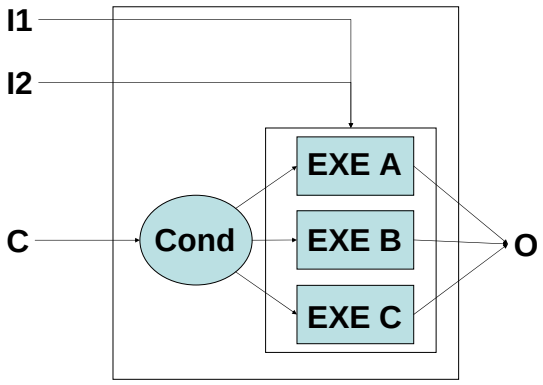

Figure 3.29: Conditional process execution in the QSDF MoC

The example of Figure 3.30 shows an example of RVC-CAL code of the PDSF explained above in Figure 3.29.

```
actor QuasiStatic() int C, int I1, int I2 ==> int O :

  cond.a: action C:[ c ] ==> guard c = 1 end
  cond.b: action C:[ c ] ==> guard c = 2 end
  cond.c: action C:[ c ] ==> guard c = 3 end

  A: action I1:[ i ] repeat 2 ==> O:[ f(i[0] + i[1]) ]
  end

  B: action I1:[ i1 ], I2:[ i2 ] ==> O:[ f(i1), f(i2) ]
  end

  C: action I2:[ i0, i1, i2, i3 ] ==>
    O:[ f(i0), f(i1), f(i2), f(i3) ]
  end

  schedule fsm cond :
    cond ( cond.a ) --> exec_a;
    cond ( cond.b ) --> exec_b;
    cond ( cond.c ) --> exec_c;
    exec_a ( A ) --> cond;
    exec_b ( B ) --> cond;
    exec_c ( C ) --> cond;
  end

end
```

Figure 3.30: RVC-CAL example of the QSDF MoC

3.4 Compilation and simulation of RVC-CAL designs

Two important tools have been proposed to manage Dataflow designs: the first tool is Open Dataflow (OpenDF) [7] for CAL edition, simulation and also for the generation of C implementations or XLIM (XML Language Independent Model) [4] representations. The second tool is OpenForge [34] for the generation of hardware implementations using the XLIM representation of OpenDF. In 2008, Wipliez et al. proposed the first tool (called Cal2C [79]) that allowed a software simulation of CAL actors by translating them to C. Later CAL2C evolved to *Orcc* [80] with new and faster compilation approaches. Moreover, this tool uses a very clear intermediate representation allowing many developers to create several code generators from RVC-CAL like C, LLVM or Promela using this IR, some transformations and pretty printer. This flexibility in the IR was the major motivation of this thesis to use Orcc for the resolution of hardware generation from RVC. In this section, we present the main compilation steps of an RVC-CAL code and the existing tools (simulators and compilers): OpenDF [7], OpenForge and Orcc. Later, we explain the limitation of

these tools concerning the hardware generation. Finally, an overview of the thesis work is presented as a solution for those limitations.

3.4.1 RVC-CAL compilation

A compiler, in definition, is a program that transforms a source code written in a source language into a semantically equivalent target language. Generally, the source language is used for describing a high level algorithm and the target language is a lower-level description used for the execution of this algorithm. Currently, compilers realize the transition from a source to a target language using the following main steps:

- syntax analysis and checking so called *code parsing*,
- generation of the intermediate representation for an easier management and control,
- analysis and optimization of the generated IR,
- creating and optimizing an abstract representation of the target language,
- printing the target language using the abstract representation and pretty printing techniques.

3.4.1.1 Code parsing

The code parsing is a conversion of a code into a structured representation. Every code respects the grammar of the language and this grammar generates two main components:

- The lexer: Recognizes the tokens defined as a set of meaningful sequences of characters, so called lexemes, that make words in the language.
- The parser: Groups the tokens into meaningful structures. The output of a parser is a representation of variables, constants, operators and statements in a tree of nodes. The most used are the AST (Abstract Syntax Tree) and the CST (Concrete Syntax Tree). The best suited for manipulating source programs is the AST. An example of AST representation is shown in Figure 3.31.

The AST is very important but not sufficient since it is very complex for optimizations and data analysis. It has also many similar forms. For example the ”FOR”, the ”WHILE” and the ”REPEAT .. UNTIL” are represented in the same way. Idem for the ”if” and the ”switch”. Moreover, some expressions have a very complex AST

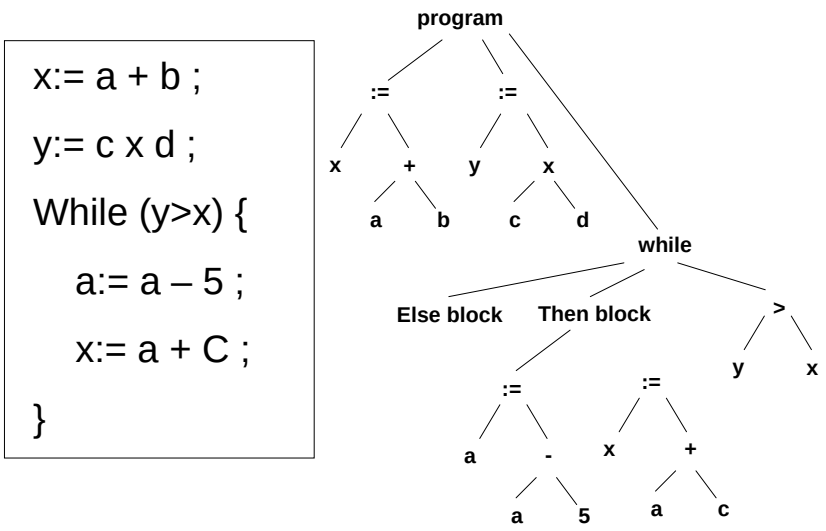

Figure 3.31: AST representation of an algorithm

representation especially for nested structures. Consequently, a simpler representation is required to explicitly represent control flow and Dataflow information. The solution is the use of Control Flow Graphs (CFG) explained below.

3.4.1.2 Control Flow Graph

A CFG is a graph composed of nodes that represent a block of statements and directed edges that represent the control flow. The same example of Figure3.31 is represented in the CFG of Figure 3.32.

This representation is very accessible and enables many optimizations such as dead code detection and removal [19], constants propagation [77], loops optimization [15] and many other optimizations detailed in the reference book of compilation called ”Dragon book” [2].

3.4.1.3 The Intermediate Representation (IR)

An AST is very important for the parsing and the analysis of the code. However, it is not able to encapsulate compilation information and other important aspects of the RVC-CAL language. The abstract tree, for example, cannot detect information about the actions order and priorities nor about tokens dependencies located in a ”guard”. This crucial information can be added to the AST but they make it very complex especially for analysis. Consequently, the notion of Intermediate

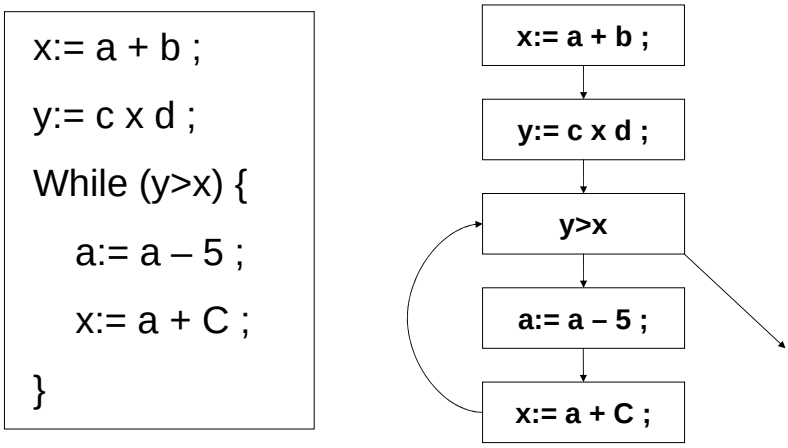

Figure 3.32: CFG representation of an algorithm

Representation is added to the new compilers. Unlike the AST, an IR is a readable data structure that contains all necessary information and details about variables, operations, FSM, priorities etc. Moreover, the IR is generally very close to the target language as the example of the Register Transfer Language (RTL) which is an IR very close to the assembly language. Currently, there are many IR used by the compilers like Low Level Virtual Machine (LLVM) [31], GIMPLE the IR of GCC, the Three Address Code or the Static Single Assignment (SSA) form which is used by the tools presented in the remaining of this thesis. The SSA representation allows easier and faster optimizations since it insures that every variable is assigned only once, and so, the constraint is single. We consider, for example, the following example of code containing a conditional statement "if" (Figure3.33):

```
1 if (condition) then
2   var := 0;
3 else
4   var := 1;
5 end
6 x := var;
```

Figure 3.33: Example of an algorithm with a conditional statement

The associated SSA form is presented in the graph of Figure 3.34. It uses a

new variable for every assignment of the variable ”VAR” creating, thus, different branches that have to be merged later when the branches are joined. For that purpose we use the ”PHI” function which considers the value of the variable that has to be used for the rest of the algorithm.

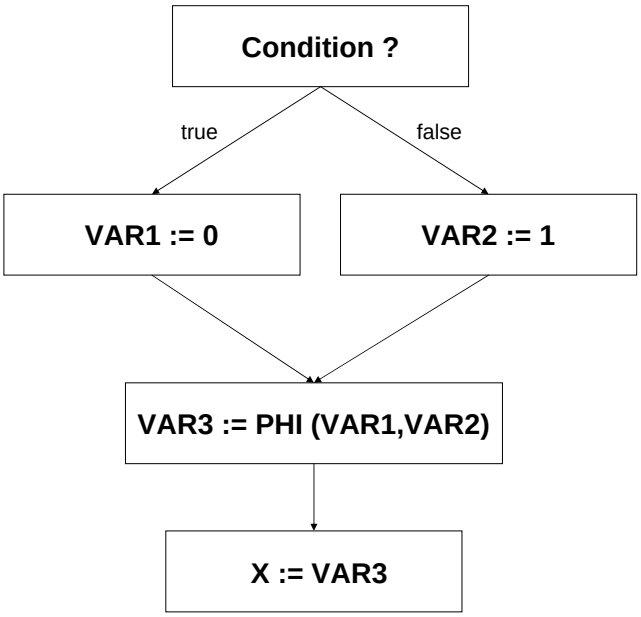

Figure 3.34: SSA representation of the example of Figure3.33

At this level, we have presented the most important notions related to the compilation of RVC-CAL programs. In the following, we present the tools used in the RVC-CAL framework for compilation and simulation.

3.4.2 Generation of HW/SW implementations with OpenDF

CAL is supported by an interpreter infrastructure able to simulate actors network. Moses [29] was the first tool that used this interpreter for the simulation of networks by featuring a graphical network editor. During the simulation it was possible to watch the evolution of states and tokens values. Because of the lack of maintenance, the project was substituted by the OpenDF environment based on the Ptolemy project [11]. In addition to the simulation, OpenDF is also able to generate an XLIM IR based on a Static Single Assignment form (SSA). In this step, the compiler parses the RVC-CAL actors and transforms them into a set of threads in the SSA form and thus creates the XLIM files for each actor. This transformation

encapsulates: code analysis, type checking, constant propagation and several other precompilation transformations.

As presented in Figure 3.35, there are two possible forms of XLIM:

- the software-oriented XLIM used by a tool called XLIM2C [67] to generate a C implementation of the design. This IR supports all RVC-CAL structures.
- the hardware-oriented XLIM which is the front-end of OpenForge which compiles this XLIM into a Verilog implementation. The hardware generator translates the SSA threads into elementary operation circuits. In addition, it manages the data exchange by creating local referees and schedulers. The SystemBuilder library is used to define the required type of FIFOs (synchronous or asynchronous). The final output of the compiler is a Verilog file for each actor and a VHDL top file of the design. However, OpenForge compiles only a subset of RVC-CAL structures. This limitation (detailed later) revealed to be the core of this thesis.

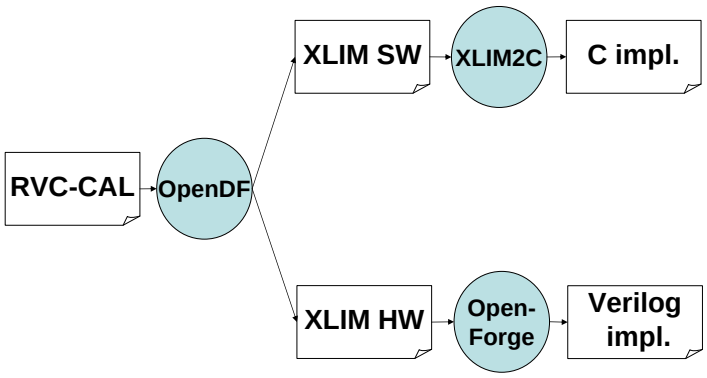

Figure 3.35: OpenDF implementations

This infrastructure was very successful in 2008 when it allowed the development and the automatic generation of software and hardware implementations of MPEG4 simple profile decoder [40]. To go further with RVC, many improvements had to be added especially for more optimizations and faster compilation. To begin, OpenDF uses the XLIM as an intermediate representation which is a very low level IR making

the generation of implementations a very long task especially for hardware. Moreover, the interpreter uses directly the AST which is different from the representation used for code generation. Moreover, the code has not been maintained for years. All these drawbacks originated the new Open RVC CAL compiler (Orcc) as a new infrastructure for the compilation of RVC-CAL. This compiler is detailed below.

3.4.3 Open RVC-CAL Compiler (Orcc)

Orcc is an integrated infrastructure for creating, editing and compiling RVC-CAL Dataflow applications. As presented in Figure 3.36, the principle of Orcc is to transform a high level RVC-CAL design into an equivalent description in a usual language (C, C++, java ...).

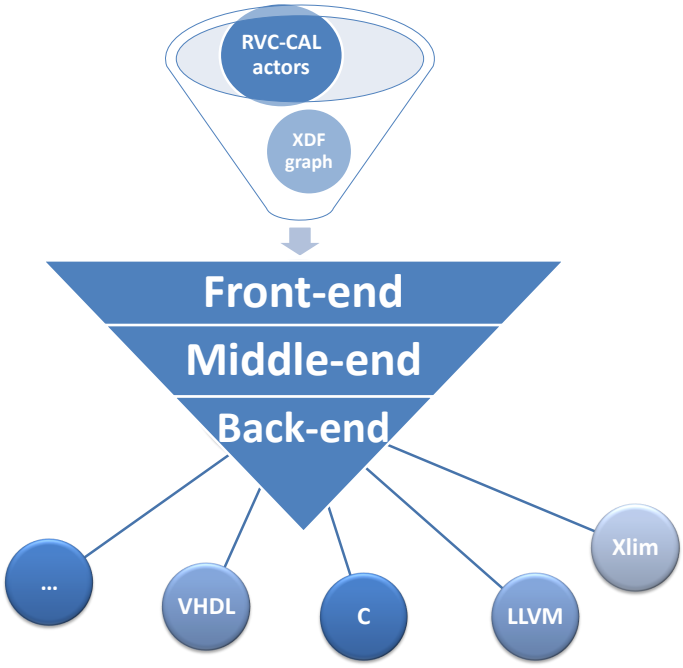

Figure 3.36: Orcc compilation flow

An RVC-CAL design is defined using XDF networks that can be managed with a network editor called Graphiti [3]. In the following, we present the compilation infrastructure of Orcc, more particularly the IR, the front-end, the middle-end and the back-ends.

[3] Available at: graphiti-editor.sf.net

3.4.3.1 The front-end

As all compilers, Orcc has a front-end that transforms RVC-CAL actors into an Intermediate Representation which is an abstract representation of all the nodes, expressions and variables of the code. Orcc uses the Eclipse Modeling Framework (EMF) to build a meta-model IR of the different structures of the RVC-CAL language. The meta-model provides a formal relationship between classes which makes the visits and the relationship analysis between any two objects easier and faster. The control of object containment is also insured by the meta-model. Thus, it is impossible that an object o contained in an object $O1$ can never be contained at the same time in an object $O2$. Moreover, the EMF provides a set of methods to move or to copy objects from a model to another one. In the following, we present the most important tasks of this front-end:

► **Code parsing** As shown in Figure 3.37, the front-end begins by parsing the actors to create the AST. This parsing is insured by an open source framework called Xtext [25]. After the specification of the grammar of a domain specif language (DSL), Xtext uses a tool called ANTLR (ANother Tool for Language Recognition) [4] to automatically generate the parser and the lexer. Moreover, the Eclipse Modeling Framework (EMF) [24] is used by Xtext to generate a meta-model of the AST which is easier to manage. Xtext facilitates the development of the compiler and the Eclipse environment of the DSL (key word coloration, syntax analysis etc.).

► **Expression evaluation** The front-end of Orcc contains an expression evaluator necessary for compile-time constants when generating the IR from the AST. The evaluator returns the compile time constant or an error if the constant is not defined for example.

► **Variables typing** After the parsing, the AST undergoes a typing step, as presented in Table 3.2, which is the transformation of types to those of the IR system. For example, the integer type is transformed into a 32-bits wide integer. This step contains also an evaluation of the expressions.

► **Type checking** Unlike the OpenDF compiler, Orcc is able to detect typing errors in the front-end. As presented in Table 3.3, a while condition for instance must be a Boolean and variable assignment has to be type compatible.

[4] Available at www.antlr.org

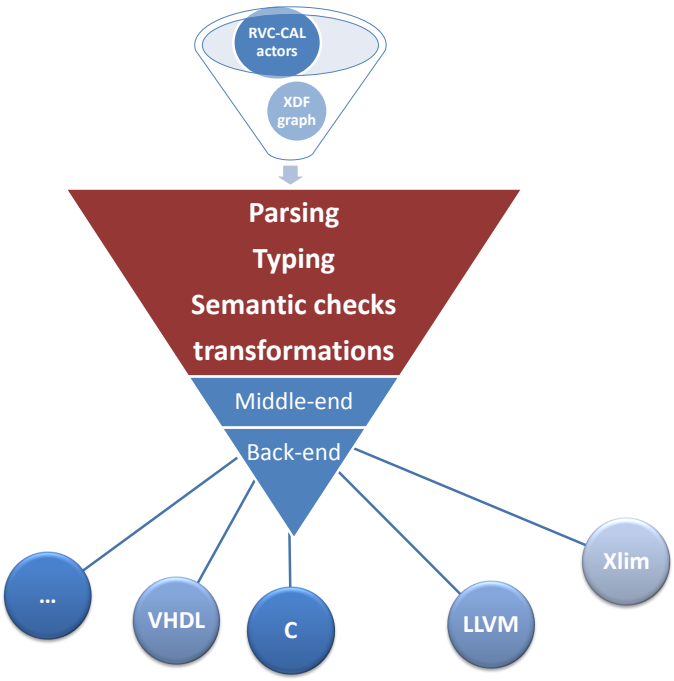

Figure 3.37: The front-end of Orcc

RVC-CAL type	Condition	IR type
`bool/float/String`		`bool/float/String`
`int/uint`		`int(size=32)/uint(size=32)`
`int(size=`e`)`	$eval(e) \in \mathbb{N}$	`int(size=`$eval(e)$`)`
`uint(size=`e`)`	$eval(e) \in \mathbb{N}$	`uint(size=`$eval(e)$`)`
`List(type=`t`, size=`e`)`	$eval(e) \in \mathbb{N}$	`List(type=`$conv(t)$`, size=`$eval(e)$`)`

Table 3.2: RVC-CAL type system conversion to IR type system

► **Scheduling and priority information** The FSM information about transitions and priorities are collected into a tag association table containing the list of actions and their associated tags. Each transformed action is referenced in this table with all the existing tags then added to the table. When all actions are added, the total order can be established from the partial order of tags with the creation of a directed graph. This graph is defined with $G = (V, E)$ where V is the set of vertices representing the tags and $E \subseteq V \times V$ is the set of edges representing the priorities. Figure 3.38 represents the directed graph of the priority $action1 > action2 > action3.1 > action3.2$.

AST node	Condition
`if`(e)	$typeof(e) =$ `bool`
`while`(e)	$typeof(e) =$ `bool`
$t := e$	$typeof(t) \cap typeof(e) \neq \emptyset$
$p(e_1, ..., e_n)$	$\forall p_i \in params(p), typeof(p_i) \cap typeof(e_i) \neq \emptyset$

Table 3.3: Type checking of AST statements.

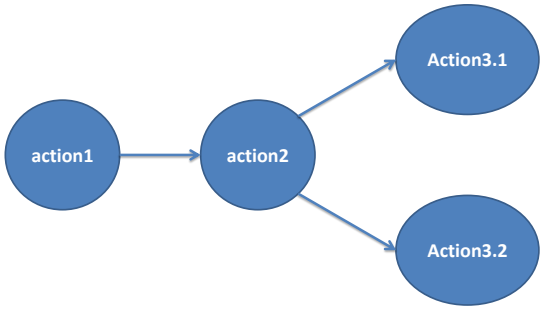

Figure 3.38: Example of priority directed graph

► **Action structure** The transformation of an action is characterized by the creation of two main parts:

- the **body** of the action that contains the processing algorithm.
- the scheduling condition so called **is_schedulable** which is a set of expressions that test the "guard" condition. The result of the **is_schedulable** procedure is a Boolean that will be used later by the scheduler.

If an action execution is dependent from a data in an input FIFO, then an input pattern is created to define the amount a required data, the name of the port and its associated local variables in the action. The required data is tested with a FIFO spy using the *Peek* statement to respect the principle of the DPN MoC.

► **Semantic transformations** The only transformation left is the statements one. All variables are transformed so that the access is exclusive to avoid conflicts of values. The For, If and While statements are transformed into a call to IR blocks called $NodeWhile$ and $NodeIf$. Idem for expressions: after the classification of the expression (unary, binary or ternary), each expression is mapped to its equivalent representation in the IR.

3.4.3.2 The middle-end

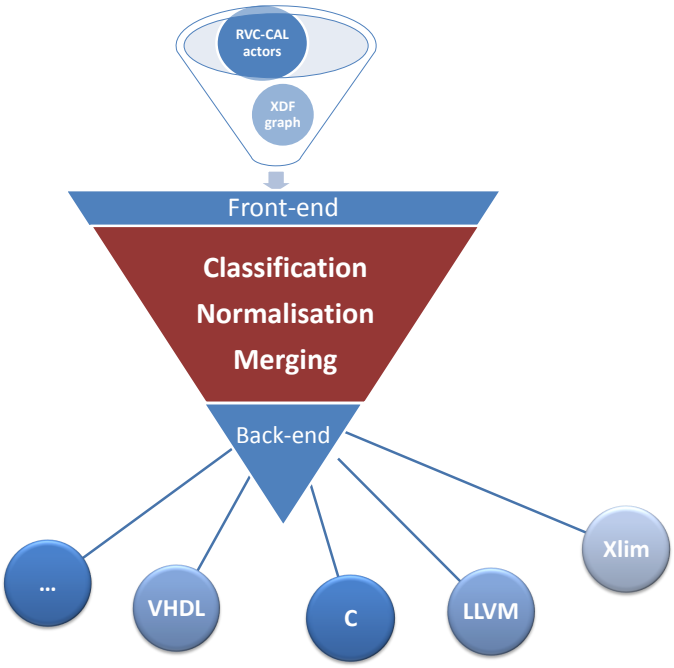

Figure 3.39: The middle-end of Orcc

After the semantic an type checking in the front-end, further analysis (Figure 3.39) can be achieved in the middle-end concerning the behavior of the actors.

The main contribution of the middle-end is the classification. The classification of the MoC of the actor (DPN,SDF,CSDF or QSDF) is important to optimize the FIFO management. As presented previously in Section 3.3.2, the SDF actor behavior can be fixed at the compilation time as they always consume and produce the same number of tokens. In the other side, the behavior of the dynamic actors and especially the time-dependent ones can never be determined at the compilation time. When the classifier detects a set of static actors, it is able to merge some of them in the same actor. This merging has a very important impact on the performance of the design especially the memory consumption and FIFO management. Before the classification, all actors are considered as dynamic ones. The scheduler has to always compute the number of present tokens in the input FIFOs and the number of empty cells in the output FIFOs. However, after the classification, the number of executions of an actor is known so the read and write processes can be replaced by data storage in tables.

3.4.3.3 Orcc back-ends

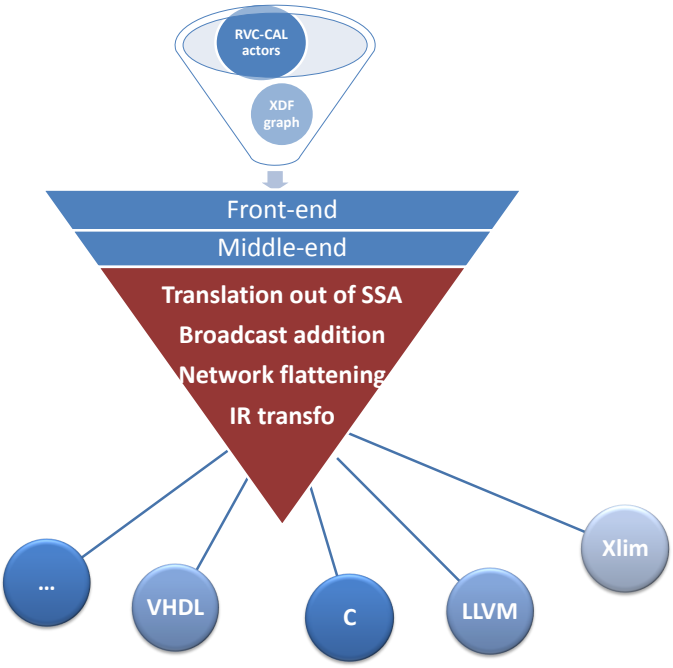

Figure 3.40: The back-ends of Orcc

The last compilation step of Figure 3.40 is the back-end ending in the generation of the target languages.

► **Existing back-ends** For a target language *X*, the generated code with Orcc is called the *X* back-end. The back-ends have many different properties depending on their abstraction level (high or low level). Consequently, every back-end has a set of specific transformations that have to be applied before code generation. Of course, it is not necessary to provide all existing languages simply because C, XLIM, VHDL and LLVM can cover over 99% of existing target architectures. Currently, many back-ends are developed in Orcc. The first one is the C back-end [79],[66], [78] as a programming reference. It has been followed by other targets like C++, VHDL, XLIM, Promela, Java and LLVM. The LLVM (Low Level Virtual Machine) is a low level language like the *Assembly* but with high level type in- formation for compiler analysis and optimization. It catches the main operations of ordinary processors and avoids machine specific constraints such as physical registers or pipelines These features of the LLVM back-end [31] make it very important for the reconfigurability

aspect of the RVC-CAL designs. Using the LLVM it is possible to load the networks on demand at the compile-time using the just-in-time (JIT) of the adaptive decoder (Jade) [32]. Indeed, LLVM is itself a low level IR that can be executed on the fly on different architectures using JIT engines. Jade generates an LLVM representation of the MPEG Video Tool Library and calls them on demand during the network instantiation and this characteristic enables the dynamic reconfiguration. Concerning the VHDL back-end [72], as we were looking for solutions to hardware generation, this back-end had the same limitations as OpenForge when it came to handling loops and multi-token reads/writes. The approach we propose in this thesis to handle multi-token data exchange and loops can be used with both the VHDL back-end and OpenForge. A tool called Synflow Studio[5] integrates a VHDL code generator that is loosely based on the former VHDL back-end from Orcc to generate hardware from DPN actors. The most important back-end for this thesis is the XLIM one. As presented previously, OpenDF generates XLIM as a specific IR which is difficult to modify. However with Orcc, it is possible to add transformations to the IR before generating the XLIM. XLIM has a lower level than the IR of Orcc and the XLIM associated transformations realize this passage to low level. We notice:

- the inlinement of functions and procedures,
- the conversion of actors to Single Static assignment form,
- the transformation of multi-dimensional list into a single dimension one,
- the initialization of lists and variables,
- ...

In Chapter 5, we present a set of transformations we added to resolved the main limitations of the hardware generation from RVC-CAL designs.

▶ **Code printing** After all the common transformations of the front-end and the middle-end, and after the target specific transformations in the back-end, the last step is to print the final code. For that purpose, Orcc uses the string template approach [59]. Unlike the programmatic approaches that require very long programs, the template approach enforces the separation between the model and the view [60]. This approach reduces significantly the redundancy bugs. Figure 3.41 presents an example of the C-back-end string template code for the constants.

[5] Available at www.synflow.com

```
Constant(constant) ::= <<
<if (constant.exprBool)><printBool(constant)><
elseif (constant.exprInt)><printInt(constant)><
elseif (constant.exprFloat)><printFloat(constant)><
elseif (constant.exprList)><printList(constant)><
elseif (constant.exprString)><printString(constant)><
elseif (constant.exprVar)><printVar(constant)><
endif>
>>
printVar(var) ::= <<
<var.use.variable.name>
>>
printBool(constant) ::= <<
<if (constant.value)>1<else>0<endif>
>>
printInt(number) ::= <<
<number>
>>
printFloat(number) ::= <<
<number>
>>
// the values of a list: {val1, val2, ..., valn}
printList(constant) ::= <<
{<constant.value: Constant(); wrap, separator=", ">}
>>
printString(constant) ::= <<
<constant>
>>
```

Figure 3.41: Constants string template code of the C back-end

3.5 Hardware compilers limitation: the multi-token case

In the previous sections, we presented the existing tools and compilation infrastructures of RVC, notably OpenDF, OpenForge and Orcc. Currently, the software generation presents very efficient implementations whether with the XLIM2C of OpenDF or the software back-ends of Orcc. On the contrary, the hardware generation presents many limitations related to especially to the reading and the writing of tokens. Indeed, RVC-CAL allows the description of actions that consume and produce more than one token per execution, which is not acceptable in hardware

because a physical memory is a set of cells that can be accessed by one or two buses once a clock period. So it is impossible to read or to write a set of data at the same time. In some specific applications where the number of exchanged data and the dynamic are small (1 to 4 bits), it is possible to consider a concatenation. However, in the video coding field, many actors present a huge amount of data so this solution is not acceptable. The only solution left is to transform an action that reads a multi-token sequence of data into a set of actions that read in a mono-token way and execute the body of the action once the necessary tokens are present. This transformation involves the addition of an FSM to properly manage this sequencing. In chapter 4, we present examples of manual transformations of RVC-CAL to get the mono-token presentation. This manual transformation, despite the fact that it is a tough task especially for complex actors, allowed us to collect much information about the final modified code to be compliant with OpenForge. In Chapter 5, we used all these information to achieve a robust automatic transformation of all RVC-CAL actors from multi-token to mono-token description.

3.6 Conclusion

This chapter, presented the fundamental part of this thesis which is the RVC standard and its associated compilations infrastructures. This standard has revolutionized the MPEG standards since it is the first standard that does not specify a coding algorithm but a way of coding. MPEG-RVC intends to replace the monolithic classic description of decoders into a set of functional units (actors) connected graphically and respecting the DPN Model of Computation. Thus, it is easily possible to reuse existing algorithms (collected into a Video Tool Library) when designing a new standard. RVC presents a domain specific language called RVC-CAL for the actors specification and also some frameworks to automatically generate hardware or software implementations like OpenDF OpenForge and recently Orcc compiler. The existing tools are able to compile all RVC-CAL actors in the software field but the hardware generation has many bottle-necks and limitations related especially to multi-token read and/or write. After the exposition of the state of the art and the localization of the problematic of this thesis, Part II presents the main contributions in terms of methodologies and techniques to resolve the problematic. The first chapter presents a methodology to accelerate the validation of hardware generation from Dataflow programs and the second details an automatic transformation inserted in the core of Orcc to resolve the limitations of hardware compilers.

Part II

Proposed Techniques And Methodologies

Chapter 4

A methodology for fast validation of RVC-CAL programs

To maximize market opportunity, a project must minimize the amount of time spent validating the hardware design without negatively impacting the quality of the validation. This validation is insured with simulations after the compilation and the synthesis of the design. If some errors occur, which is often the case, designers have to debug the circuit by checking the time evolution of all signals and memory states. This debugging is a very long and tough task. Since RVC-CAL is target agnostic, we proposed a methodology that allows a maximum of functional validation in a software platform before testing in the hardware one. This methodology is first based on the generation of a software implementation of the RVC-CAL code. Then this implementation is rapidly tested, corrected and validated using a test software platform. Once the design is valid, the new correct RVC-CAL code can be used for hardware generation.

In the following, we present the new global method for functional validation and assessment of an RVC-CAL code.

4.1 Fast validation approach principle

As presented in Figure 4.1, the design algorithm is described at a high level with RVC-CAL language. Orcc is used to generate a software implementation of the algorithm using its C language generator. We chose the C language because it is the most known and consequently the easiest to manage and debug. Then a software platform is used for functional validation and FIFO sizing. Once the code is correct, it undergoes a modification to be synthesizable with a hardware compiler, called OpenForge, by unrolling the loops and the repeat structures. The validation of this code is realized with the same software platform. The implementation is finally insured using a hardware synthesis and prototyping platform.

4.1.1 Existing validation methods

As seen before, RVC-CAL code used originally to be validated with the OpenDF simulator. This plug-in simulates the behavior of an RVC-CAL design and displays any required information using the ”println” command. The stimulus stream has to be manually written using another actor as presented in Figure 4.2.

We can have in the display window a trace of the ”println” command such as:

data 0 = 4
data 1 = 11
...
data 9 = 19

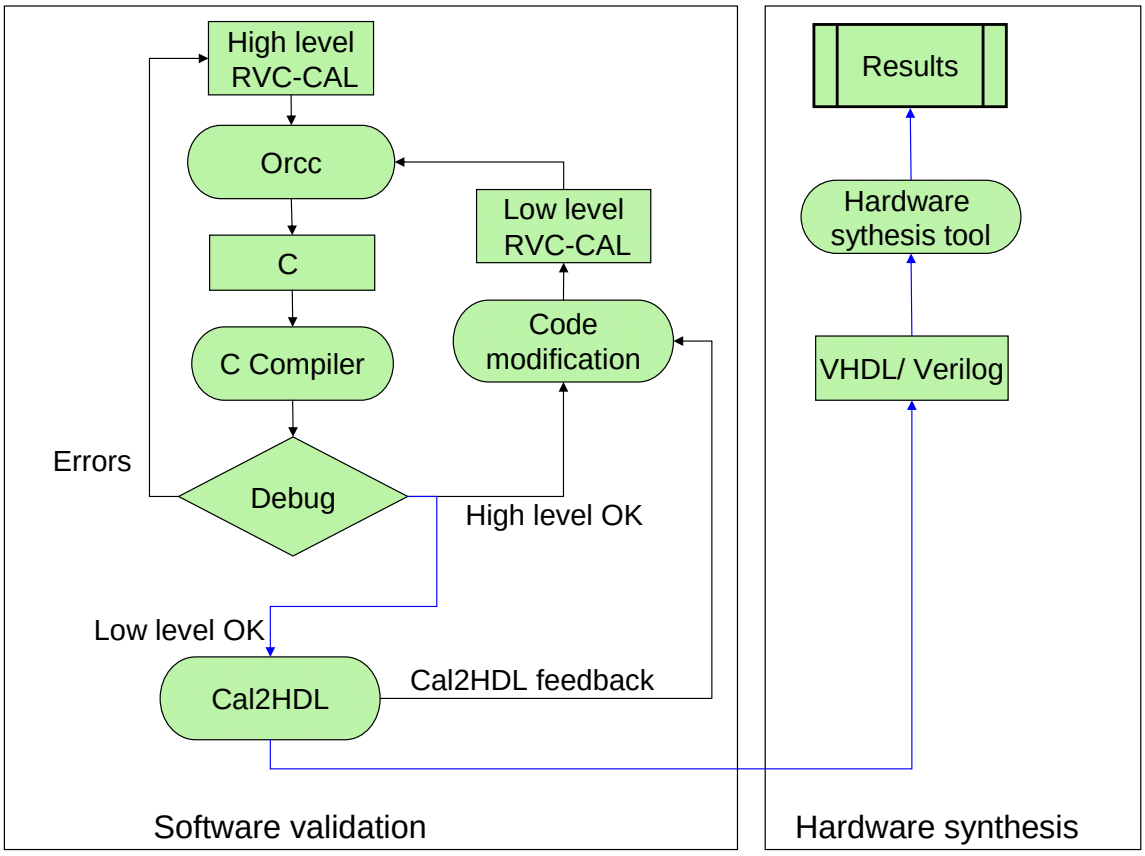

Figure 4.1: Method overview

```
actor data () ==> int(size=13) OUT :

List(type: int(size=9), size=10) data := [ 4, 11, 4, 2, 16, 15, 8, 3, 11, 19];

  int inc := 0;

  int out := 0;

  increment: action ==> OUT:[ out ]
  guard
    inc < 10
  do
    out := data[inc];
    println("data" + inc + "=" + data[inc]);
    inc := inc + 1;
  end

end
```

Figure 4.2: Stimulus data actor example

It is also possible to check the evolution of FSM states, variable value, table content etc. Moreover, the simulator elaborates and parses the actors and the associated graph providing a feedback about connection errors like type mismatching in the case of a connection between two ports with different types. It can also detect dead states in an FSM or an uninitialized variable. This simulator remains satisfying for simple actors, but drawbacks appear when testing a multi-port actor with a specific scheduling, or for instance with actors that have to test a huge amount of data to be validated. Effectively, as presented in Figure 4.2 the stimulus actor is itself an additional algorithm to be developed. In case of multi-port actors to test, we have to prepare the required stimulus lists and to write correctly the data actor. Consequently it is, paradoxically, equivalent to validating a validation platform! In addition, for a great amount of data to test, large tables have to be created with thousands of values that have to be correctly written in a manual way which is a difficult task. Finally, there is no automatic comparison solution for output data, so it is impossible to validate for example the over 100.000 pixels of a processed CIF image. Considering the video processing context, we noted the drawback of using such simulator and the necessity to find new solutions able to handle both of complexity and huge data amount of the validation process. Below, we present the advantages of using Orcc compiler as a solution for the limitations of OpenDF simulator. The use of Orcc for a preliminary validation of the RVC-CAL design before hardware generation represents the originality of the approach [45], [46].

4.1.2 Functional validation in a software platform

As introduced above, the novelty of this work is the use of Orcc for a maximum validation steps in a software context. For an RVC-CAL design, Orcc is able to gen-

erate a C code for each actor and a top file that manages these actors. An important library of functions necessary for the compilation is developed and contains very useful functions especially for scheduling, threads management and FIFO management. The library contains also a set of functions that allow reading/writing data from/into images and videos files, comparing output data with a given correct file, and displaying the resulting data. Considering the fact that we dispose of such potent tool and libraries, we present in the following the main steps of the new validation process.

▶ **Step1: High-level development and software generation** As presented in Figure 4.3, the objective is to get a fast validation of the algorithm using the simplest description way which is the high level one.

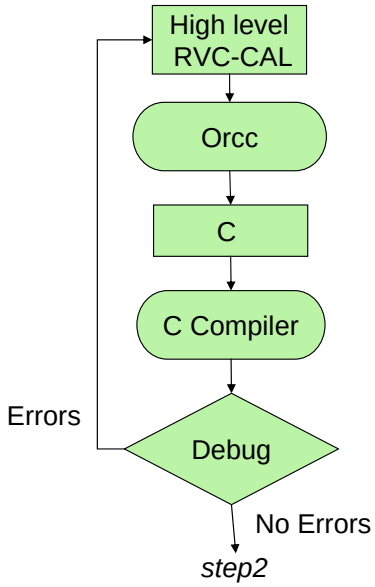

Figure 4.3: Step1: High-level validation

For example, to write 16 tokens in an output port, we use the high level structure of Figure 4.4 rather than directly using the more complex low level structure of Figure 4.5.

```
1 write: action ==> OUT:[tab] repeat 16
2 end
```

Figure 4.4: High level RVC-CAL example

```
write: action ==> OUT:[out]
do
  out := tab[counter];
  counter := counter + 1;
end

write_done: action ==>
guard
  counter = 16
do
        counter := 0;
end

schedule fsm write:
  write (write     ) --> write;
  write (write_done) --> nextState;
...
end
```

Figure 4.5: Low level RVC-CAL example

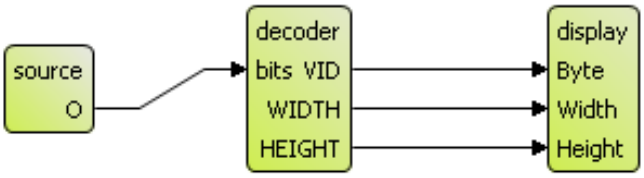

Figure 4.6: Source and display connection for MPEG4 decoder design

Once the design is completed, we add the source and the display actors already proposed in the VTL of MPEG RVC (see Figure 4.6).

Orcc is launched using the C backend of the list of Figure 4.7 and generates the C files for each actor. The generated C files of the source and the display call the required functions of the Orcc library later in the compilation step.

Finally, a solution is built and a C compiler is used to compile and debug this solution. We use configuration instructions to specify the input video file that will be handled with Orcc library functions. The output video can also be compared with a given correct one. If the algorithm is correct, we can see a display of the required video (Figure 4.8). Otherwise (Figure 4.9), we debug the solution. Of course, the software debug is very less-time consuming than hardware one. Once the RVC-CAL code is checked, so that it generates a correct software implementation, the first validation step is achieved. Some optimizations can be added since Orcc allows the specification of FIFO sizes. Therefore, it is interesting to look for the minimal FIFO size that keeps a correct video display, as this size has a direct impact on the memory consumption of the intended hardware implementation. As Orcc applied the same size on all FIFOs of the design, it is possible to use Graphiti to set the size of a specific FIFO. At this level, the RVC-CAL errors are easier detected and faster corrected.

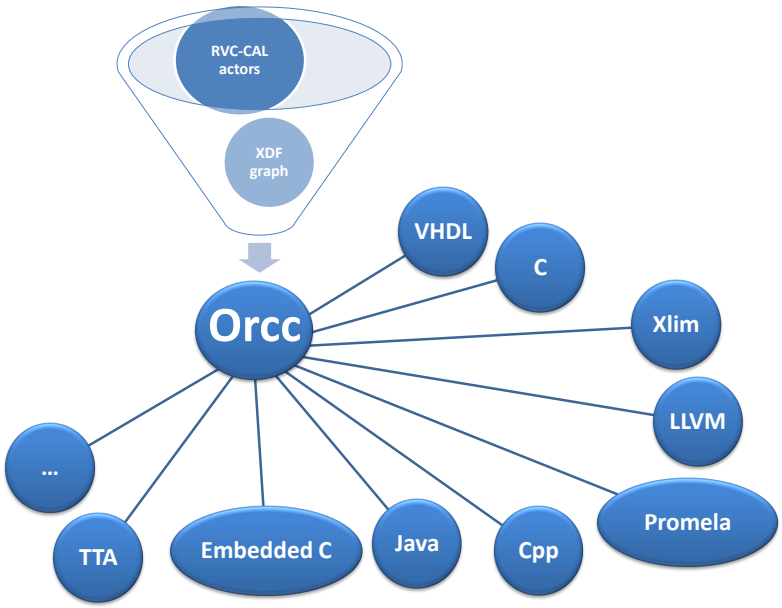

Figure 4.7: Orcc list of backends

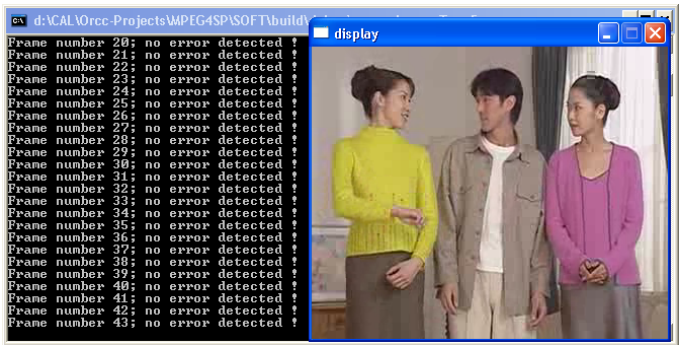

Figure 4.8: Correct video display

► **Step2: From high to low level** This description step presented in Figure 4.10 contains the most important change in the RVC-CAL code. As explained in Section 3.5, each actor of the design, already validated in a high level description, has to be changed into low level to suit the transformation process of hardware compilers. An explosion of the number of code lines is possible.

As we can see in the examples of Figures 4.4 and 4.5, for a multi-token write on only one port, the number of code lines increases from 2 to 18. This difference generally implies errors since there are additional actions, states and variables associated with algorithms and tests on counters. These additional algorithms have to be tested to check if the behavior of the actor is still correct. Concerning the FIFO

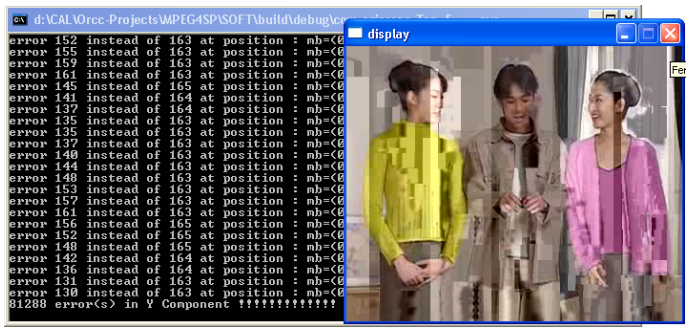

Figure 4.9: Wrong video display

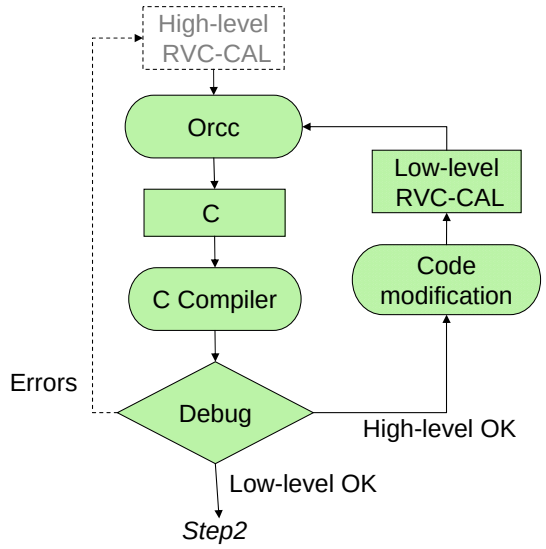

Figure 4.10: Step2: Low-level validation

size, it is possible to have further optimizations since in the mono-token case, data are stored into internal buffers and all the firing rules verify the existence of only one token in the FIFO. Therefore, all FIFO sizes may be set to the value ”1”. Nevertheless, there exist some cases for which it is necessary to set other values than ”1” and it concerns especially the broadcast FIFOs. Actually, a FIFO may be broadcast to many ports that belong to different actors and, as presented previously in the Dataflow MoC, actors are independent from each other so they have different processing cadences. Consequently, if an actor execution is too much faster than another one while they are sharing the same FIFO, the internal buffer of the slower actor will be full at a certain time so it is necessary to increase the FIFO size of this actor to avoid the blocking of the network.

This second step is achieved when the RVC-CAL code of the mono-token case generates a software implementation that behaves correctly and produces the re-

quired data stream. The obtained design is now ready for hardware generation.

▶ **Step3: Hardware generation** Writing and validating a low level RVC-CAL in the software platform is a very important step since it guarantees that the global algorithm is correct. However, there may be other source of errors while generating hardware implementations. These errors are especially related to types and dynamics of the different variables. Actually, in the software platform, all types of integer with a dynamic between 2 and 16 bits are set to short (i16) and those between 17 and 32 bits are set to integer (i32). Consequently all type-related errors are automatically avoided but this is not the same case for hardware compilers. In the algorithm example of Figure 4.11, the variable ”i” is defined as a signed variable with a dynamic range of 8 bits involving a maximum value of ”127” while the algorithm uses a loop expecting ”i” to reach the value of 600. In this case, the software implementation is going to behave correctly but a hardware compiler will detect a bit-width problem since it would synthesize a register with the same type of ”i” (8 bits) to contain the comparison value of 600 which is not supported by this register.

```
1 int (size=8) i;
2
3 example: action ==>
4   foreach i in 0 .. 600 do
5     //(instructions)
6   end
7 end
```

Figure 4.11: Type-error algorithm example

A wrong dynamic may also induce a value change and errors. This may happen for example when assigning a token value from a variable of 8 bits type to a variable of 7 bits type. In this case the most significant bit (MSB) is lost. So if the variable is signed, the value will be inversed and if it is unsigned the value is divided by ”2”. The previous case can scarcely happen but it cannot be detected during the hardware generation. It can be considered as a last source of errors that can only be detected and debugged in the hardware simulation step.

Practically now, we have a set of low level RVC-CAL actors connected in a directed graph and validated in the previous steps. The hardware generator considered is OpenForge as presented in Chapter 2, and the front-end of this compiler is the XLIM format. For recall, OpenDF is the tool that compiles RVC-CAL actors to output a back-end in the XLIM format. For hardware generation, this XLIM is the front-end of the hardware compiler OpenForge (see Figure 4.12). The XLIM format is a set of Static Single Assignment threads transformed by the compiler into an intermediate format called SLIM which is a template language whose goal is reduce the syntax to the essential parts without becoming cryptic. The SLIM is then transformed into circuits based on basic operators. The final generation outputs a Verilog

file for each actor and a VHDL top file that connects the ports of these actors, so considered as components, with the adequate FIFOs (synchronous or asynchronous). Since OpenForge is developed by Xilinx Inc Company, developers created all the functions and components required for defining the FIFO architectures into a special library called SystemBuilder. This library contains the entity definition and the behavior of many useful VHDL entities: RAM (RAM_bool, RAM_int, RAM_dualclock), FIFO controllers, FIFOs (synchronous FIFO, asynchronous FIFO). We use the generated Verilog files, the top file and the SystemBuilder libraries to make projects for hardware simulation. For this step, stimulus files called "test bench" are required. In the Section 4.2, we present an original method for automatic generation of both stimulus files and test benches using Orcc.

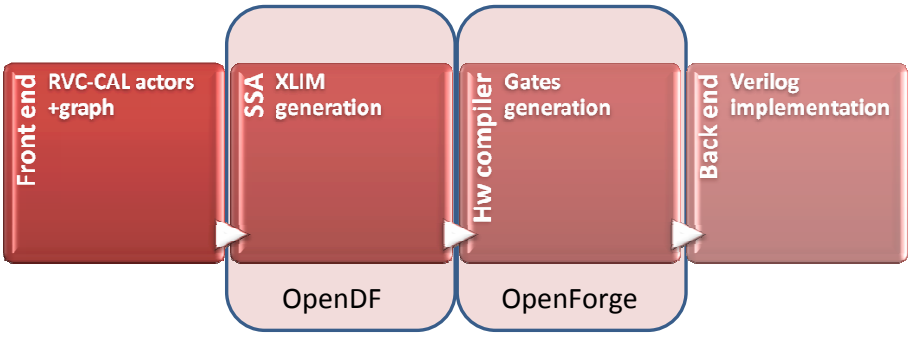

Figure 4.12: OpenForge compilation steps

Another novelty of using Orcc is the fact that we can generate an XLIM code using an associated back-end and this XLIM is compatible with OpenForge. We explained in 3.4.2 the reasons of dropping out the SSA generator of OpenDF in the favor of the XLIM generator of Orcc.

4.2 Automatic generation of test benches and stimulus files

In the hardware simulation field, a test bench is a special HDL file that calls the Top file of the design to be tested as a component. In the test bench, we define the clock frequency and the Offset duration. For each input or output port, the test bench defines an associated signal. These signals are used for input ports to set stimulus values and for output ports to display them in a time line graph containing the evolution of each signal at any specific time of the simulation process. The stimulus is set using two methods:

- **Direct assignment inside the code**

```
signal <= 'value0';
wait for period ;
signal <= 'value1';
etc ...
```

This solution is used for simple simulation cases when the number of ports and stimulus values is reduced.

- **Data reading from files**

In case of complex components with a large number of token to be tested, we use stimulus files since it is possible for the test bench file to read values from some text format files using the following instructions:

```
file sim_file_in  : text is "decoder_in.txt"; -- file definition
readline(sim_file_in, line_number); -- pointing at a given line number
read(line_number, signal); -- value assigning to a stimulus signal
```

Concerning the output signals, in the case of a simple test without an important number of data to check, it is possible to simply check the evolution of the signal in the simulation environment. Otherwise, the test bench is able to read a reference file and to automatically compare all the generated values with those of the reference using the instructions:

```
file sim_file_in  : text is "decoder_out.txt"; -- file definition
readline(sim_file_out, line_number); -- pointing at a given line number
read(line_number, out_signal); -- value assigning to a stimulus signal

assert (out_signal = test_signal) -- condition to check
report "on port out incorrectly value computed :
'test_signal' instead of : 'out_signal'" -- report if condition is false
severity error;
```

Since the format of a test bench is predefined, it is possible to generate an automatic file for each actor or network of an RVC-CAL design. The required information is the names and the types of all ports which is simply accessible in the Intermediate Representation of Orcc. Therefore, we developed pretty printing files called ”String templates” that allow using compiler IR and simple algorithms to automatically write a correct test bench.

At this level, only one thing is missing: the automatic generation of the stimulus files. For that purpose, Orcc presents an interesting option while generation the C implementation which is the ”trace” option. This functionality adds lines to define and fill a new file with all data values that passed through a FIFO. Every port is now connected to a test file that is going to save values during the software simulation step. Consequently, after a correct software simulation we can find the trace files and use them later for hardware generation.

4.3 Pipeline methods

In [45, 46],we present further optimization solutions can be added using a ping-pong data management algorithm [14]. The principle of this algorithm is to avoid the repetitive latency caused by data storage, while reading tokens, by combining the reading and the writing of the tokens in the same action. The idea is to first write the input data in the half part of a memory, and then to use this data while writing in the other part. Finally it only consists on switching the reading and the writing pointers in opposite from a half memory part to the other. An example of ping-pong memory management of a 4 buffers memory is presented in Figure 4.13.

In RVC-CAL language, the solution is to use pointer functions such as *ra()* and *wa()* in the example of figure 4.14. After reading and filling the half of the memory, a Boolean flag *half* changes the read and the write pointers represented by ra() and wa(). Thus, an alternation of read and write is created in action *gradient*. Consequently, while writing tokens, that actor is reading new ones for the next process which decreases considerably the processing time.

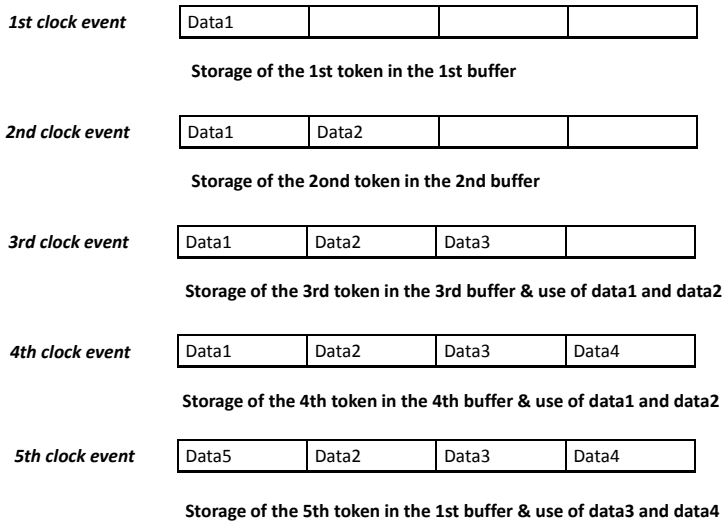

Figure 4.13: Ping pong example of a 4-buffer size memory management

Using such algorithms has considerably increased the performances. We notice that for some actors, we obtained a throughput every clock period. Tests have shown later that OpenForge generates a BRAM for internal buffers with two R/W buses. So for actions presenting more than two memory accesses, it is logically impossible to have the results in only one clock period. Another case causes the delay when, in the same instruction of the body of an action, an operator is repeated several times. To run this instruction process in only one clock period, OpenForge has to synthesize a hardware block for each operation. If the number of operators exceeds the synthesis constraint of "performance VS area" then the compiler cannot generate all the required hardware blocks Thus the instruction cannot process in only one clock period. We can go further in the optimization of the pipelined design to solve the cases above. Indeed, it is possible to anticipate the computation and create more internal buffers for customized pipeline, but this way of designing architectures is contradictory to the motivation of using MPEG-RVC standard. The purpose is still to write in high level and to reach a valid implementation the fastest possible. So such deep optimizations make the design very low level and as complex as writing directly in HDL.

Once the design is transformed to low level, validated, optimized and compiled to hardware it is necessary to simulate the hardware for the final validation. In the following, we present an automatic method for generating test benches and stimulus vectors using Orcc pretty printings and generation options.

```
1
2   bool half := false;
3   function wa() --> int :
4     bitor( bitand(cnt,BLK_SZ*BLK_SZ-1),
5     if half then BLK_SZ*BLK_SZ else 0 end )
6   end
7
8   function ra() --> int :
9     bitor( bitand(cnt,BLK_SZ*BLK_SZ-1),
10    if half then 0 else BLK_SZ*BLK_SZ end )
11  end
12
13  read_bloc_size : action BLK_SZ_IN:[size_in] ==>
14  do
15   sz_in[cnt]:=size_in;
16   cnt:=cnt+1;
17  end
18
19  done : action MAX:[max], MIN:[min] ==>
20  guard cnt= BLK_SZ*BLK_SZ
21  do
22  max_tmp:=max;
23  min_tmp:=min;
24  half := not half;
25  cnt:=0;
26  end
27
28  gradient : action BLK_SZ_IN:[size_in] ==> BLK_SZ_OUT:[out]
29  var
30    int out
31  do
32    sz_in[wa()] := size_in;
33    if (max_tmp - min_tmp<GRADSTEP) then
34      out:=BLK_SZ;
35    else
36      out:=sz_in[ra()];
37    end
38    cnt:=cnt+1;
39  end
40
41 schedule fsm read :
42  read(read_bloc_size)--> read;
43  read(done)          --> read_write;
44  read_write(gradient)--> read_write;
45  read_write(done)    --> read_write;
46 end
47
48 priority
49  done > read_bloc_size;
50  done > gradient;
51 end
```

Figure 4.14: Ping-Pong memory management example

4.4 Comparison with manual flow

The RVC methodology enables a very important gain in development time since an RVC-CAL code is dedicated to the internal behavior of the actor and all communication issues are managed automatically by the libraries of compilers. Moreover, this

standard enables an easy reuse of existing functional units which decreases even more the development time of new designs. Currently, with our methodology, we reduce also the validation time which is an important step. To test the impact of our work, we applied the method while developing a baseline of the LAR codec (detailed in Chapter 7). The development, validation and test of the LAR took about 2,5 month-man while in direct VHDL it took over 8 month-man. It is very important to mention that over 90% of the conception time was achieved in the open source software platform where the debug and the validation are easier and faster.

4.5 Conclusion

This chapter presented an overview of the hardware generation flow from RVC-CAL designs. The classic generation and validation method is based, in a first step, on the use of OpenDF for the edition of low level RVC-CAL code, the simulation of the code, and the generation of the XLIM intermediate representation then, in a second step, the use of OpenForge for the transformation of the SSA threads into low level HDL implementations. The originality of our work resides in the introduction of a functional methodology that allows multi-level validation of the design using a software platform. This methodology consists of using the Open RVC-CAL Complier tool to generate a software code in every level of RVC-CAL descriptions. This code is compiled and simulated with many videos. At the low level RVC-CAL description, this validation ensures that the algorithms are correct and generally the hardware generated files are correctly implemented. In some cases, the hardware circuits present errors but they are not related to the algorithm. Most of them are resolved with a good FIFO sizes corrections. This global framework introducing a software functional checking before the synthesis process is significantly faster than a hardware implementation directly from the RVC-CAL description.

The most time-consuming part of the flow remains the manual transformation of the RVC-CAL from high to low level. This can be explained by the fact the code is longer and consequently harder to debug because of the inaccurate feedback of OpenForge. This problem may be resolved by improving OpenForge Java source code, but the very complex source of OpenForge and the low level of the XLIM Intermediate representation make this task almost impossible. The solution is to use the intermediate representation of Orcc to generate an XLIM compatible with OpenForge and to add automatic transformations to the intermediate representation of Orcc while generating the XLIM. The next chapter presents a set of automatic transformations introduced in the IR of Orcc during the XLIM generation that transforms the high level features into low level ones allowing the hardware generation with OpenForge.

Chapter 5

Automatic hardware generation from RVC-CAL

5.1 Introduction

The hardware conception flow from RVC-CAL designs consists of two main parts: the CAL development and the hardware generation using compilers. As presented previously, hardware compilers have the limitation that they cannot compile multi-token features of the language. Indeed, a processor is preconceived to automatically manage sequential multi-read or multi-write from a memory. However, in a hardware implementation, we have to explicitly define this sequencing using a state machine as presented in Chapter 4. This task is very time-consuming if it is done manually and represents the main bottleneck of the conception flow. Consequently, the proposed solution was to insert an automatic transformation in the compiler core to detect the non-compliant features and make the necessary changes [43].

This chapter presents the main contribution of our work which is an automatic transformation that transforms high level features of RVC-CAL into low-level equivalent structures that are compliant with hardware compilers.

5.2 Localization of the automatic transformation

As introduced above, the idea of an automatic transformation in the core of an RVC-CAL compiler is the best solution for a general and target agnostic transformation. Currently, the best existing hardware compiler for RVC-CAL is OpenForge (so called Cal2HDL). As a reminder, the compilation process has two main processing steps: the XLIM generation using OpenDF and the Hardware generation with OpenForge.

The hardware generator translates the SSA threads of the XLIM file into elementary operation circuits. In addition, it manages the data exchange by creating local referees and schedulers. The SystemBuilder library is used to define the required type of FIFO (synchronous or asynchronous). The final output of the compiler is a Verilog file for each actor and a VHDL top file of the design. The limitation of OpenForge is that it cannot compile multi-token rules which are omnipresent in most actors since they make the high level development easier and faster. Our work focuses on how to automatically transform the data consumption from multi-token to mono-token while preserving the same actor behavior.
The drawback of adding transformation directly to OpenForge is the fact that its IR is very low-level and its interpreter uses directly the AST which has a different scope from code generators one. Consequently, we adopted Orcc which is a new RVC-CAL compiler that offers a clear IR and visitor design pattern. The motivation behind this choice is also because there are ongoing works on XLIM back-end in Orcc [4]. Therefore, the SSA generator of OpenForge can be replaced by the Xlim generator of Orcc and all required transformations are developed in the core of Orcc. The resulting conception flow is summarized in Figure 5.1.

After the localization of the transformation in the conception flow, it is impor-

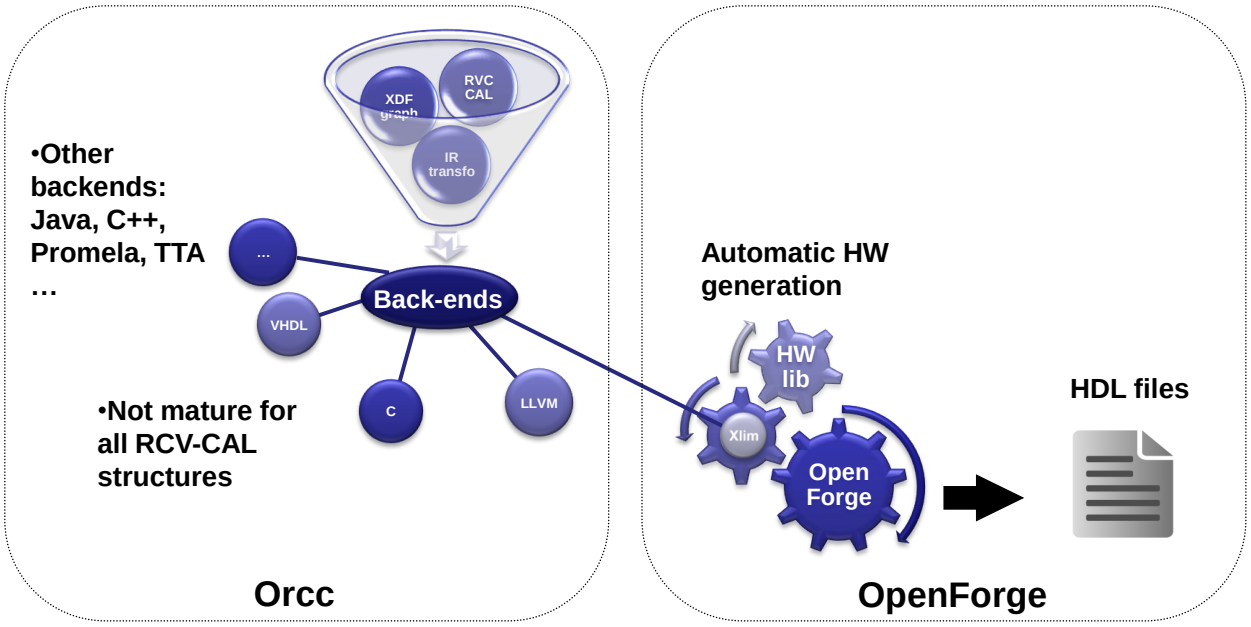

Figure 5.1: New conception flow with Orcc

tant to introduce the mathematical formalism of the model of computation before detailing the transformation process. In the following section, we present the most important mathematical notions and equations related to the model of computation, and we will rely on them to better explain the transformation mechanism.

5.3 Actor behavior

A presented in Section 3, RVC-CAL language is the transformation of the Dataflow Process Networks (DPN) model of computation into an executable or implementable language. This model of computation is based on the exchange of data tokens between functional entities called *actors*. The only information that fires an actor is the presence of enough data to satisfy one of its firing rules. Once a rule is satisfied, a corresponding local function called *action* is executed by consuming tokens from the input FIFO and producing others on the output FIFO.

A Dataflow actor is defined with a pair $< f, R >$ such as:
* $f : \mathbb{S}^m \rightarrow \mathbb{S}^n$ is the firing function,
* $R \subset \mathbb{S}^m$ are the firing rules,
* For all $r \in R$, $f(r)$ is finite.

An actor defines a set of firing rules that precise the necessary conditions that make this actor consume and produce tokens. An action is fireable or schedulable

iff : the execution is possible in the current state of the FSM (if an FSM exists), there are enough tokens in the input FIFO, and a guard condition returns true. An action may be included in a finite state machine or untagged making it higher priority than FSM actions.

The FSM transition system of an actor is defined with $< \sigma_0, \Sigma, \tau, \prec >$ where Σ is the set of all the states of the actor, σ_0 is the initial state, $\prec$ is a priority relation and $\tau \subseteq \Sigma \times \mathbb{S}^m \times \mathbb{S}^n \times \Sigma$ is the set of all possible transitions. A transition from a state σ to a state σ' with a consumption of sequence $s \in \mathbb{S}^m$ and a produced sequence $s' \in \mathbb{S}^n$ is defined with (σ, s, s', σ') and denoted:

$$\sigma \xrightarrow[\tau]{s \mapsto s'} \sigma' \qquad (1)$$

To solve the problem of the existence of more than one possible transition in the same state, RVC-CAL introduced the notion of priority relation such as for the transitions $t_0, t_1 \in \tau$, t_0 a higher priority than t_1 is written $t_0 \succ t_1$. A transition $\sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'$ is enabled iff:

$$\neg \exists \sigma \xrightarrow[\tau]{p \mapsto q} \sigma'' \in \tau : p \in \mathbb{S} \wedge \sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'' \succ \sigma \xrightarrow[\tau]{s \mapsto s'} \sigma' \qquad (2)$$

5.4 Transformation overview

Let us consider an actor with a multi-token firing rule:

$$r \in \mathbb{S}^k \text{ such as } |r| = [r_0, r_1, .., r_{k-1}]$$

This rule fires a multi-token action a realizing the transition $source \xrightarrow[\tau]{a} target$ and $\mathbb{I}$ the set of all input ports. The idea of the intended transformation is to separate the input and output patterns from the action and create mono-token actions that use these patterns. In the following, we detail the required variables and actions to be added and we explain the way this transformation copes with the all existing cases of finite state machines.

5.4.1 Actions and variable creation

▶ **Multi-token read transformation** The transformation creates for every input port an internal buffer to store data and behave as a local FIFO. To manage the data transfer in a FIFO, it is necessary to create counters called indexes for reading and writing. Thus, every time we write n data in the FIFO the write index increments $(IdxWrite := IdxWrite + n)$ and ditto for the read index when a set of data

is definitely consumed. Consequently and as presented in Figure 5.2, at any given time, $buffer[IdxWrite]$ is the last input, $buffer[IdxRead]$ is the first input and the difference $(IdxWrite - IdxRead)$ represents the number of available tokens in the FIFO. The token $buffer[IdxRead - 1]$ is the last used token. It exists physically in the FIFO but it cannot be used anymore.

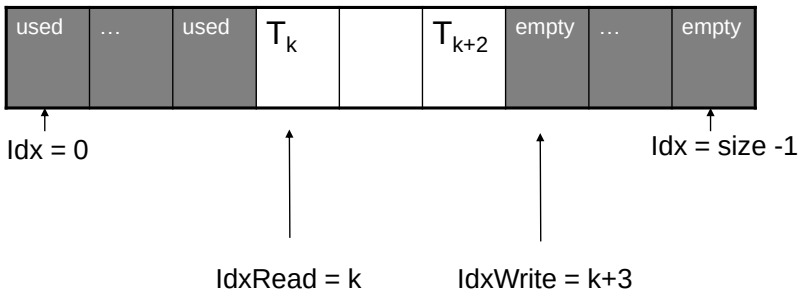

Figure 5.2: Example of buffer indexes

We explain later the importance of this index difference in the schedulability of the created actions. The indexation presented above is enough for the unlimited FIFO case which is not realizable especially for hardware implementations. So what happens when $(IdxWrite = Size - 1)$? To solve this problem, we implemented a circular FIFO. This type of FIFO is based on the idea that data contained in the registers from buffer[0] until buffer[IdxRead -1] are used and so they can be erased and overwritten by new data. Thus, the token received when the buffer is full can simply be placed in the first cell of the buffer $(buffer[0])$. At the same time, it would be an error if we set the value of the write index to zero because the difference $(IdxWrite - IdxRead)$, that represents the number of available tokens, will be negative. As shown in Figure 5.3, the solution is to use the modulo operator with the indexes and thus when $IdxWrite = size$, we have $(IdxWrite \bmod size) = 0$. For an easier mask computation, we use a power of 2 buffer size. If the number of tokens to read in the multi-token rule is not a power of 2, the size will be the closest one. Therefore, the *bitand* operator is used instead of the *modulo* one.

Actually, data storage in the created buffers have to be mono-token. So a new action is created for that purpose. This action has to be independent from the process actions, and then we consider the untagged actions since they are outside the FSM. Such action is defined in Figure 5.4. The input pattern of the port is now located in the untagged action and an important *guard* condition is added to check the availability of at least one available cell in the buffer. In fact, as the buffer is circular, it is very important to verify that $(IdxWrite - IdxRead) < size$. To resume, the transformation creates for every input port an internal buffer with read and write indexes and clips r into a set $\mathbb{R}$ of k firing rules so that:

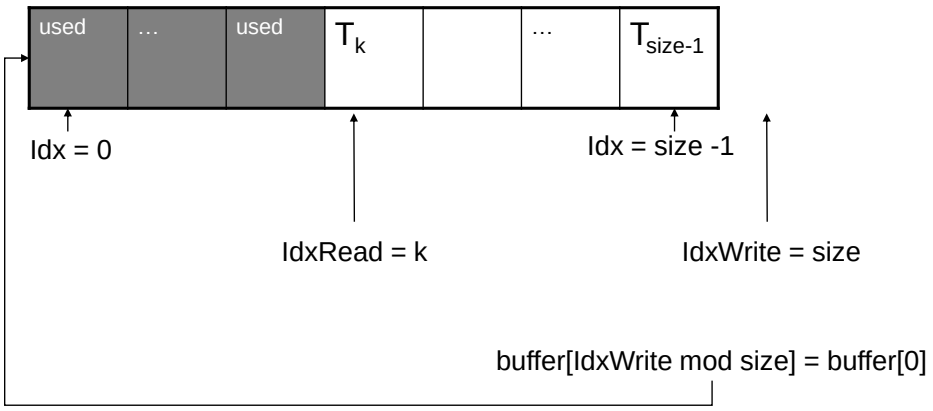

Figure 5.3: Circular FIFO management

$$\forall i \in \mathbb{I}, \exists! \rho \in \mathbb{R} : \begin{cases} \rho : \mathbb{S}^1 \rightarrow \mathbb{S}^0 \\ |r| = 1 \\ g_\rho : IdxWrite_i - IdxRead_i \leq size_i \end{cases} \quad (3)$$

with ρ a mono-token firing rule of an untagged action $untagged_i$, g_ρ is the guard of ρ and $size_i$ the size of the associated internal buffer defined as the closest power of 2 of r_i.

```
action PORT:[ token ] ==> // untagged action
guard
  IdxWrite - IdxRead < size
  // condition that at least one cell is erasable
do
  buffer[IdxWrite & (size-1)] := token ;
  // masked read index
  IdxWrite := IdxWrite + 1 ;
end
```

Figure 5.4: Untagged action for mono-token read

► **Actions transformation** For each port $i \in \mathbb{I}$, an untagged action is created associated with read/write indexes and a buffer. The indexes and the buffer are created as global variables so they can be used by other actions. Indeed, if the actor contains another firing rule (multi-token or mono-token) that uses a port which is

already connected to a buffer, when this rule fires it has to consume the tokens in the buffer and not those in the FIFO. Consequently, all input patterns related to this port are deleted and replaced by a *read* instruction from the associated buffer. Let us consider a firing rule $r \in \mathbb{S}^2$ with a *guard* condition on the first port with a positive token. We suppose $|r|$=[1,2]. This rule is defined by:

$$\begin{cases} g : [x] | x > 0 \\ r = [t_0 \in g, [t_1, t_2]] \end{cases} \quad (4)$$

After creating the untagged actions for both ports, we have two buffers ($buffer1$ and $buffer2$) with their indexes ($IdxRead1$, $IdxRead2$, $IdxWrite1$, $IdxWrite2$). This rule has to be transformed into $r\star$ such as:

$$\begin{cases} g1 : IdxWrite1 - IdxRead1 \geq 1 \\ g2 : IdxWrite2 - IdxRead2 \geq 2 \\ g3 : buffer1[IdxRead1] > 0 \\ r = [\lambda, \lambda] \end{cases} \quad (5)$$

where $g1$ and $g2$ are the conditions that there are enough data in the buffers to fire the action and g3 is the translation of the original guard g of r in the new created context.

Concretely, if we consider the basic example of action sum-5 of Section 3.2, the transformation outputs the code of Figure 5.5.

▶ **Multi-token write transformation** An action may have the multi-token structures in the input pattern and/or the output pattern. While reading data, there is no problem of skipping the FSM to fire an untagged action that reads a token from the FIFO. However, creating untagged actions for writing data involves several problems since the data transfer from an actor to the FIFO has to be completely done after the execution of the body of the transformed action. Therefore, if this action restarts before the drain of the output buffer, then some tokens will be crushed. The solution is to add a mono-token write action for each port managed by a new FSM branch. In this paragraph, we explain the created FSM macro-block separately and in Section 5.4.2 we detail the way this branch is inserted depending on the existing actor scheduling.

Analogically to the multi-token read transformation, the output pattern is removed from the action and affected to a mono-token-write one. This mono-token action requires a buffer to store data. This buffer is not shared by other processes so we can use a single index that increments while draining data and resets at the end. While writing tokens another firing rule may be validated and causes the firing of an unwanted action. To avoid the non-determinism of such a case, we use an FSM to put the actor in a writing loop so it can only write tokens. In a general case, we

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 add: action IN:[ i ] repeat 5
 ==> OUT:[ s ]
 var
   int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + i[k] ;
   end
 end
end
```

(a) High-level algorithm

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 List (type: int (size=8), size = 8) buffer;
 // size = 8 is the closest_power_of_2(5)
 int IdxWrite := 0;
 int IdxRead := 0;

 action IN:[ i ] ==>
 guard
   IdxWrite - IdxRead < 8
 do
   buffer[IdxWrite & 7] := i ;
   IdxWrite := IdxWrite + 1 ;
 end

 process: action ==> OUT:[ s ]
 guard
        IdxWrite - IdxRead > 5
 var
  int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + buffer[k + (IdxRead&7)] ;
   end
   IdxRead := IdxRead + 5;
   // update IdxRead
 end

end
```

(b) automatically generated low-level algorithm

Figure 5.5: Automatic transformation of the Sum-5 actor

suppose that the multi-token action to transform realizes the transition:

$t_{existing} = source\_state \xrightarrow[\tau]{action} target\_state$ of Figure 5.6

The new FSM loop is entered using the transition:

$$t_{write} = source\_state \xrightarrow[\tau]{transformed\_action} write\_state.$$

The write action loops in the *write_state* with the transition:

$$t_{loop} = write\_state \xrightarrow[\tau]{write} write\_state.$$

When all necessary tokens are written, the loop is exited using a *write_done* action and a transition to the *target_state* defined with:

$$t_{end\_loop} = write\_state \xrightarrow[\tau]{write\_done} target\_state \succ t_{loop}.$$

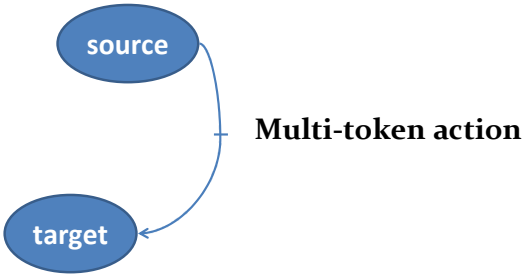

Figure 5.6: Existing FSM transition of the transformed action

For example, the actor A of Figure 5.7 is defined with

$$f : \mathbb{S}^0 \rightarrow \mathbb{S}^2$$

```
actor A () ==> int OUT1, int OUT2:
   a: action
   ==>
   OUT1:[out1], OUT2:[out2] repeat 2
   do
      {treatment}
   end
end
```

Figure 5.7: RVC-CAL code of actor A

The transformation creates the FSM macro-block of Figure 5.8.
The actor of Figure 5.7 is then transformed as shown in Figure 5.9.

5.4.2 FSM creation cases

We consider an example of an actor defined as:

$$f : \mathbb{S}^3 \rightarrow \mathbb{S}^2$$

containing the actions $a1..a5$ such as $a3$ is the only action applying a multi-token

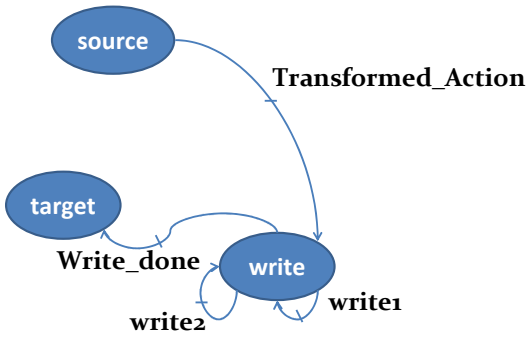

Figure 5.8: Created FSM macro-block

firing rule $r \in S^3$. If there is no multi-token structure in the output pattern then the FSM is not going to be changed since the untagged actions related to the multi-token read are added outside FSM. In the case of multi-token write, the previously explained FSM macro-block has to be inserted. Two cases may be encountered:

► **The actor does not have an FSM** In this situation, all the actions are untagged. Consequently, the fact of simply adding the FSM branch is not correct because this macro-block is a substitution of the transformed action and the existence of untagged actions ($a1, a2, a4, a5$) will distort the priority order of the original action.

The solution is to create an initial state containing all the actions and add the created FSM macro-block of $a3$ (previously presented in Figure 5.8). The resulting FSM is presented in Figure 5.10 where:
$\forall i \in [1..5], \exists! \tau_i : \tau_i = init\_state \xrightarrow[\tau]{a_i} init\_state.$

Once all actions are set into an FSM, it is possible to add the transformed action and its associated FSM branch without disturbing the behavior of the actor.

► **The actor has already an FSM** We now suppose the same actor scheduled with an initial FSM as shown in Figure 5.11.

In this case there is no need to create an initial state because there is no risk of losing a priority or any other behavior specification. Therefore, the transition

```
actor A () ==> int OUT1, int OUT2:

  int Idx1 := 0;
  int Idx2 := 0;
  List(type:int, size=1}buffer1;
  List(type:int, size=2}buffer2;

  a: action
  ==>
  // output pattern deleted
  do
    {treatment}
  end

  write1: action ==> OUT1:[out1]
  var
        int out1
  do
    out1 := buffer1[Idx1];
    Idx1 := Idx1 + 1;
  end

  write2: action ==> OUT2:[out2]
  var
        int out2
  do
    out2 := buffer2[Idx2];
    Idx2 := Idx2 + 1;
  end

  write_done: action ==>
  guard
        Idx1 = 1,
        Idx2 = 2
        //check data is fully drained
  do
    Idx1 := 0;
    Idx2 := 0;
    //reset counters
  end

  schedule fsm source_state:
    source_state(a)          --> write_state;
    write_state(write1)      --> write_state;
    write_state(write2)      --> write_state;
    write_state(write_done) --> target_state;
  end

  priority
        write_done > write1;
        write_done > write2;
  end

end
```

Figure 5.9: RVC-CAL code of transformed multi-token write

$t = S1 \xrightarrow[\tau]{s \mapsto s'} S2$ is substituted with the FSM macro-block of $a3$ as shown in Figure 5.12.

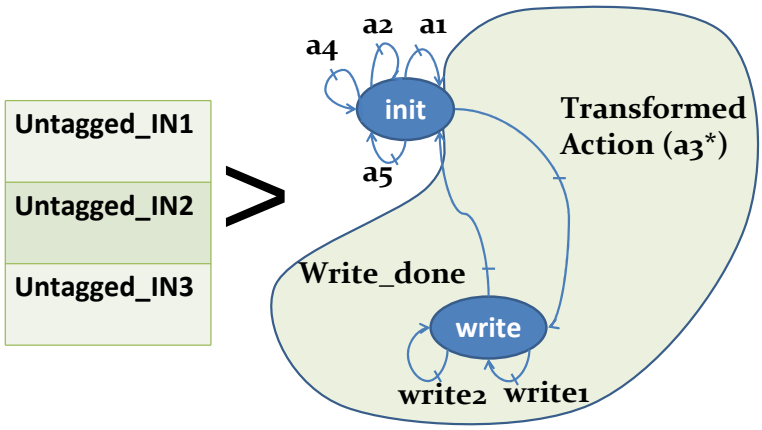

Figure 5.10: FSM with created initial state

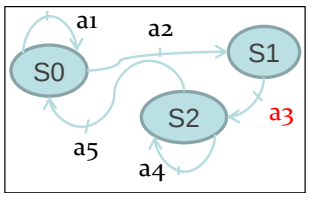

Figure 5.11: Initial FSM of an actor

This section presented a general transformation that can be applied on any RVC-CAL actor. The transformation detects the multi-token patterns of the actor and automatically substitutes them with a set of actions. All necessary states, transitions, state variables and buffers are automatically created. In some cases, the actor may have multi-token read and/or multi-token write but it is not very complicated to undergo a general transformation. In the next Section, we present some optimization solutions for these actors.

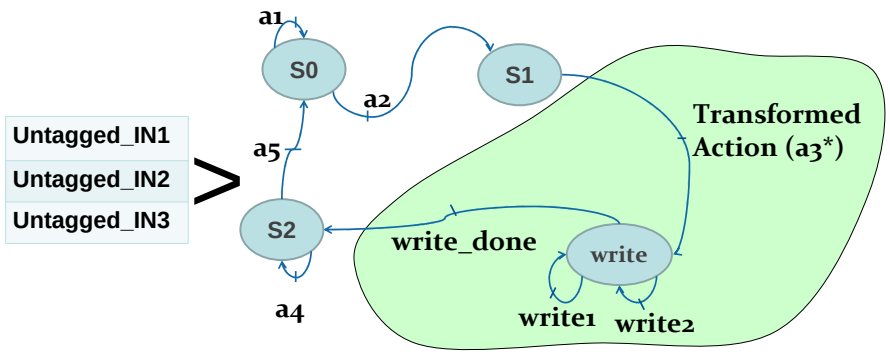

Figure 5.12: Resulting FSM transformation

5.5 Transformation steps and optimizations

To improve the transformation, some optimization solutions were added [47]. In the previously presented transformation method, we used the untagged actions to store data in the internal buffers. Consequently, all buffer accesses for reading and writing have to use masks (modulo or bitand operators). These masks involve more logic components and obviously more latency than a direct memory access. The crucial point of using internal buffers is the *guard* condition on FIFO tokens. Indeed, let us consider the example of this action:

```
1 a: action INPUT:[i] repeat 10 ==>
2 guard
3         i[9] > 0
4 do
5         {instructions}
6 end
```

This action fires when there are at least 10 tokens in FIFO and the 10th token is positive. If we use a mono-token read action that stores data in a specific buffer of this action, then we create a problem of lost data if the condition ($i[9] > 0$) is not true. In fact, the tokens not used by action a are not any more in the FIFO and if another action will fire using the same port it will use tokens of the FIFO while it had to use those in the buffer of action a. The optimization step is consequently based on an actor analysis. Data conflict cases are avoided if there is no *guard* checking FIFO tokens values, or if actions do not share ports. If such cases are detected it is possible to use *read* and *read_done* actions exactly like explained above in Section 5.4.1 for the

multi-token write case.

Let us reconsider the basic example of the "clip" actor of Figrue 5.13(a). The optimized transformation of this actor is presented in Figure 5.13(b). It is notable that the two indexes of the circular buffer are replaced by a simple counter and thus, all masking operations are avoided.
In a general case of an actor defined with: $f : \mathbb{S}^3 \rightarrow \mathbb{S}^2$ the created FSM macro-block is shown in Figure 5.14.
It is possible to go further in the actor analysis and remark that the process of any token is independent from the others. Indeed, to clip a token value, the only required information is that value and none of the other 255. The optimal transformation is consequently a simple mono-token action that reads a token, clips the value and writes the result (see Figure 5.15).

5.6 Validation and Miscellaneous transformations

To test and validate the automatic transformation, it was necessary to continue using the approach of Chapter 4 which means that the validation is performed first in a software platform before moving to the hardware one. We considered existing designs of several decoders already presented in the VTL of MPEG-RVC and we applied the transformation using the C back-end of Orcc. The transformation performed correctly for MPEG4 simple profile decoder (see Chapter 6), MPEG AVC decoder and JPEG codec. It was also successfully applied on some newly developed RVC-CAL designs of still image codecs: the accordion-JPEG [42] and the LAR [23] detailed later in Chapter 7. After the software validation step, the hardware generation is applied using OpenForge and we faced two major hardware-generation-specific problems:

▶ **The local array**

Variables of type array can never be synthesized to hardware if they are defined locally in an action. Indeed, the hardware generators consider a memory block for each HDL component and not for each function of a component. Since every actor is transformed into a component, all memories used by its actions have to be defined outside the actions procedures. In addition, every memory is managed by a *memory controller* that induces an important area consumption. That is why hardware compiler may assign several lists to a single memory block with a single memory controller as an optimization step. Therefore, we developed a new transformation that visits all the actor local variables, detects the presence of type *list*, removes the variable and creates a new global one with the same type. Of course, all uses and definitions of the list in the SSA form are set to the new variables.

```
actor clip () int (size=9) IN
==> int(size=9) OUT:

 clipper: action IN:[ i ] repeat 256
 ==> OUT:[ i ] repeat 256
 do
   foreach int k in 0 .. 255 do
     if (i[k]<0) then i[k]:=0;
     else
       if (i[k]>255) then i[k]:=255;
       end;
     end;
   end
 end
end
```

(a) Clip actor CAL code

```
actor clip () int (size=9) IN
==> int(size=9) OUT:

 List (type: int (size=9), size = 256) data;
 int counter :=0 ;

 read: action IN:[ i ] ==>
 do
   data[counter] := i ;
   counter := counter + 1 ;
 end

 done: action ==>
 guard
   counter = 256
 do
   counter := 0 ;
 end

 process: action ==>
 do
   foreach int k in 0 .. 255 do
     if (data[k]<0) then data[k]:=0;
     else
       if (data[k]>255) then data[k]:=255;
       end;
     end;
   end
 end

 write: action ==> OUT:[out]
 var
   int out := 0;
 do
   out := data[counter] ;
   counter := counter + 1 ;
 end

 schedule fsm state0:
   state0 (read)    --> state0;
   state0 (done)    --> state1;
   state1 (process) --> state2;
   state2 (write)   --> state2;
   state2 (done)    --> state0;
 end

 priority
   done > read;
   done > write;
 end

end
```

(b) Optimized CAL code

Figure 5.13: Original and optimized clip actor

► **The integer division**

The division operation is one of the major bottle-necks of hardware generation. It is recommended for hardware developers to find solutions while writing an algo-

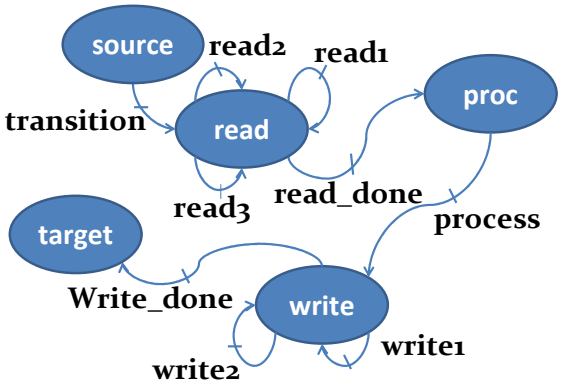

Figure 5.14: FSM created macro-block for optimal transformation

```
actor clip () int (size=9) IN
==> int(size=9) OUT:

 clipper: action IN:[ i ]
 ==> OUT:[ i ]
 do
   if (i<0) then i:=0;
   else
     if (i>255) then i:=255;
     end;
   end;
 end
end
```

Figure 5.15: Further optimization of the clip actor

rithm to avoid the divisions and use shifts instead. However, in RVC-CAL context, such specification cannot be considered because the language aims at writing in a target agnostic way. However, we can substitute an integer division by an equivalent function that uses Euclidean division which is a set a repetitive basic operations applied on all the bits of the dividend and the divisor as explained in the following algorithm example of (*short*/*short*) Euclidean division:

```
short divSS(short num, short den)
     {
         short result = 0;
         int i;
         /* If true, then the result must be negative. */
         int flipResult = 0;
         short remainder;
         int denom;
```

```
    int mask;
    int numer;
    if (num < 0)
    {
        num = -num;
        flipResult ^= 1;
    }
    if (den < 0)
    {
        den = -den;
        flipResult ^= 1;
    }
    remainder = num;
    /* Cast the denominator to a long so that MIN_INT looks like
       a positive number.(We need 33 bits to represent -(MIN_INT)).
    */
    denom = ((int)den) & 0x0000FFFF;
    mask = 0x8000;
    for (i=0; i < 16; i++)
    {
        /* Cast the numerator to a long so that -(MIN_INT) appears as
           a positive value (we need 33 bits to represent it). */
        numer = ((int)(((int)remainder & 0x0000ffff) >> (15 - i)));
        if (numer >= denom)
        {
            result |= mask;
            remainder = (remainder - (den << (15 - i)));
        }
        mask = (mask >> 1) & 0x7fffffff;
    }
    /* If the signs of the inputs did not agree, then make the result
       negative. */
    if (flipResult != 0){result = -result;}
    return result;
}
```

The algorithm above has been implemented in a *DIV* function and a new transformation is developed to detect division operations and replace them with this function as follows:

```
1  a: action ==>
2  do
3   result := dividend/divisor;
4  end
```

is transformed to:

```
1  function DIV (int (size=16) dividend, int (size=16) divisor) --> int (size=16) result:
2  {Euclidean division algorithm}
3  end
4  a: action ==>
5  do
6   result := DIV(dividend,divisor);
7  end
```

This transformation resolves the problem of integer divisions but it creates a considerable latency in the process that cannot be avoided. All these implementation details are later discussed in Chapter 6.

5.7 Written code reduction

Another very important aspect of the transformation is the time saved for not writing long codes. The multi-token structures are omnipresent in advanced video coders and their presence in an action multiplies the number of code lines by 8 times for multi-token write (Figures 5.7 and 5.9) and over 2 times for multi-token read. Generally a factor of 7 is often noted as presented in the following table 5.1 that contains a comparison of code lines numbers of some developed actors.

	High-level	Low-level
DPCM	74	523
Management blocks 2x2	27	167
Max 2x2	8	50

Table 5.1: Code lines Comparison

5.8 Conclusion

In this chapter, we introduced a general automatic transformation of RVC-CAL algorithms from high level to low-level. This transformation creates the necessary actions, states and variables to make a new algorithm that contains only mono-token firing rules with a similar global behavior of the original actor. This transformation is applied in the core of Orcc compiler and makes changes directly in the intermediate representation. This aspect of the transformation localization in the conception flow is very important since it can be applied on any back-end of Orcc. This property proved to be crucial for the validation of the transformation itself. Indeed, even if

the resulting changes in the IR are intended for hardware generation, it was possible to use a software platform to validate the transformation. To perform correctly, we added miscellaneous transformations to manage some hardware specific problems namely the local lists and the integer division. It is noticeable that actor analysis can lead to optimize the transformation of some actors that contain multi-token firing rules but they are enough complex to undergo the general transformation.

The whole methodology functional validation and hardware generation introduced in Chapter 4 associated with the transformation detailed in this Chapter for high level transformation, is applied on the MPEG 4 Simple Profile video decoder and the LAR still image codec. In the next Chapters, we present the design of the two applications platforms, we expose the implementation results and we discuss the differences with reference implementations.

Part III

Experiments And Results

Chapter 6

Technological solutions of MPEG-RVC decoders

To assess the performances of the previously presented transformations, we started a functional validation in a software platform of some RVC-CAL designs such as MPEG-4 part 2 simple profile Decoder, MPEG-4 part 10 FREXT design, JPEG etc. For the hardware implementation, we synthesized the design of MPEG-4 part 2 SP. This choice is explained by the fact that it has a stable design in the Video Tool Library (VTL). It also includes various image processing algorithms with more or less complexity. Moreover, the VTL contains a reference hardware oriented design (presented below) which is very important to assess the importance of this work. In the following we present an overview of the MPEG-4 part 2 and part 10 decoders architectures already developed in the Video Tool Library of MPEG-C. We also present the implementation results of the MPEG-4 part 2 Simple Profile and a comparison with an academic high level synthesis tool called GAUT [18],[17].

6.1 MPEG-4 part 2 Simple Profile

MPEG decoders have almost a common design. It starts with a parser that extracts motion compensation and texture reconstruction data from the bitstream generated by the Huffman encoding. The parser is then followed by reconstruction blocks for texture/motion and a merger. In the following, we present the main existing RVC-CAL designs of MPEG-4 part 2 Simple Profile decoder.

6.1.1 The hardware oriented architecture

This is a low level architecture designed by Xilinx for automatic hardware generation with OpenForge. All actors have mono-token firing rules and some algorithms are pipelined. For example, the IDCT2D is decomposed into a set of 12 actors such as rowsort, transpose, retranspose etc. This pipelined architecture is presented in Figure 6.1.

The IDCT1D is also split into pipelined actors (scale, combine, shuffle ...) as presented in Figure 6.2.

This decoder design generated the fastest hardware implementation using RVC-CAL. Consequently, it will be considered as a reference for the next implementations using the automatic transformation of Chapter 5. In [39], Jörn Janneck presented the implementation comparison results of Table 6.1 between two hardware implementations of the MPEG-4 part 2. The first one is the generated implementation from the Xilinx design and the second one is a manual directly written VHDL (http://www.xilinx.com/bvdocs/ipcenter/data_sheet/ds520_prod_brf.pdf). It is noticeable that the 26 BRAMs, a 16 bytes memory block with two input and output ports, of the VHDL IP are limited to 4-CIF image size while the 22 BRAMs of

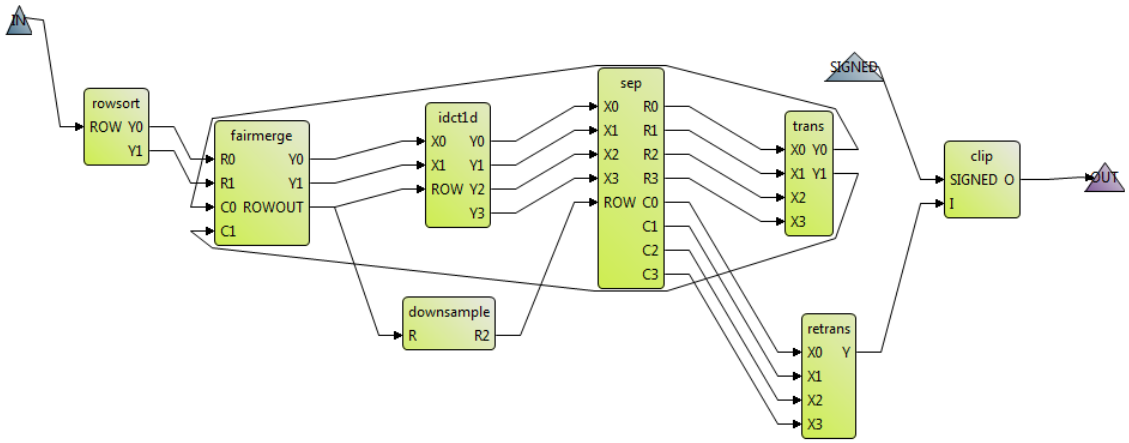

Figure 6.1: Pipelined architecture of the IDCT2D

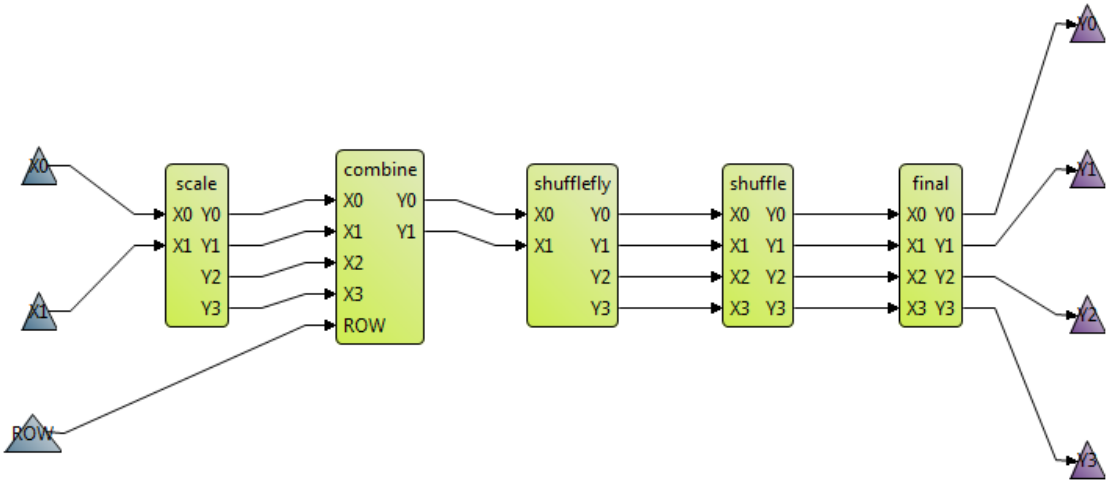

Figure 6.2: Pipelined architecture of the IDCT1D

the hardware generated from RVC-CAL support HD image size and they can be reduced to 16 for 4-image CIF size. The VHDL IP performs only a 4-CIF image size, with 180K macroblocks/s at 100MHz. However, the CAL decoder manages HD image size and performs 243K macroblocks/s at 120 MHz.

Criterion	VHDL IP	CAL
Slice	4637	3872
LUT	7923	7720
FF	2637	3576
BRAM	26	22
Mult	34	7
Lines of code	15000	4000

Table 6.1: MPEG-4 part 2 SP implementation comparison: CAL VS VHDL

6.1.2 Parallel architecture

Figure 6.3 shows one of the first architectures for MPEG-4 part 2 SP, designed in a parallel way by the EPFL of Lausanne and optimized by Ericsson [1] company during "ACTORS" project. It is characterized by a parallelization of the decoding of Y, Cb and Cr.

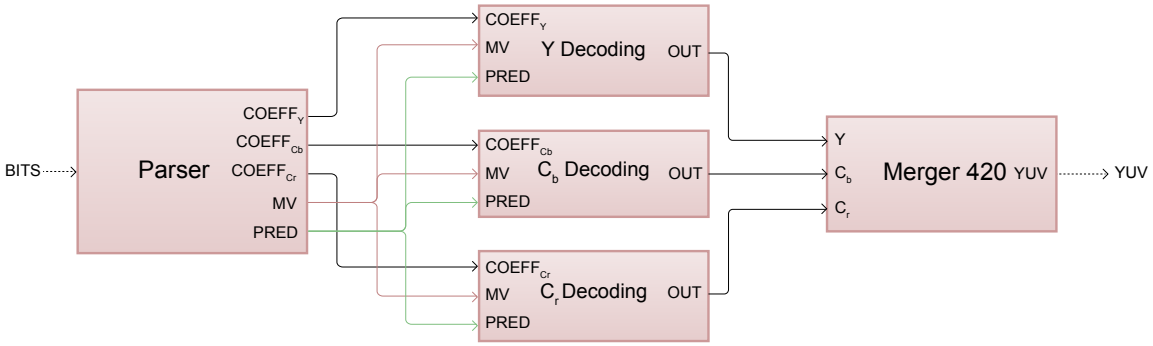

Figure 6.3: MPEG-4 part 2 Simple Profile architecture

The simple profile is a full example of decoding techniques that encapsulates predictions, scan, quantization, IDCT transform, buffering, interpolation, merging and especially the very complex step of parsing. An actor may be instantiated more than one time, so for 27 FU there are 42 actor instantiations. The detailed subnetwork of the synoptic graph of Figure 6.3 is presented in Figure 6.4.

[1] www.ericsson.com

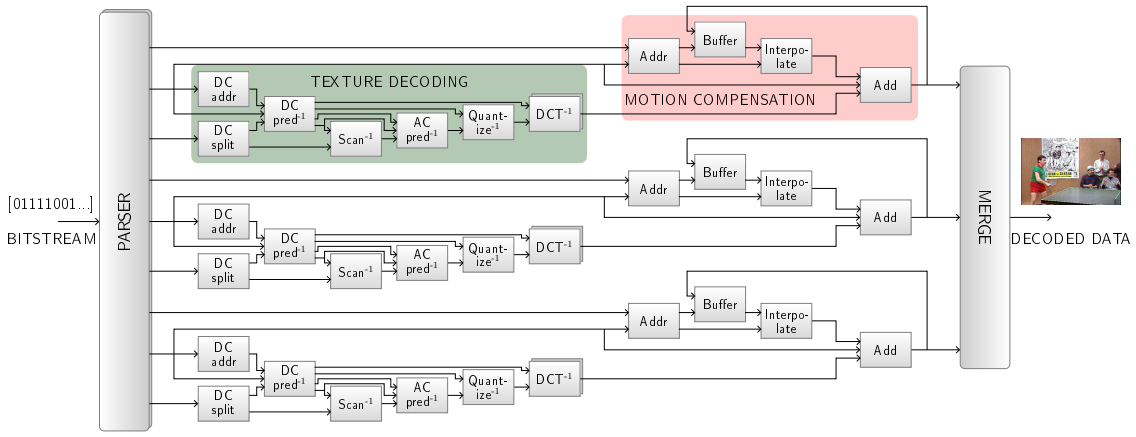

Figure 6.4: MPEG-4 part 2 SP detailed architecture

► **The parser** The parser is the actor that understands the grammar of the compressed data (bitstream) and is responsible of interpreting and extracting the necessary information for the next actors. This information represents a stream of control tokens, block type information and Video Object Planes (VOP) which are the elementary units of the compressed video with MPEG4. The bitstream is sequentially decoded, explaining the very long code of the corresponding FSM. The behavior of the parser is considered as the most complex of the whole decoder. Table 6.2 gives an idea about the complexity of parsers in MPEG 4 Simple Profile and MPEG Advanced Video Coding (AVC).

	Actors	Levels	Parser size kSLOC	Decoder size kSLOC
MPEG-4 SP	27	3	9.6	2.9

Table 6.2: Composition of MPEG-4 Simple Profile RVC-CAL description

► **The texture decoder** This part, detailed in Figure 6.5, is responsible of the intra decoding which is the inverse of the frequency domain transformation (DCT). It is composed of:

- a DC/AC splitter,
- the addressing: an actor that computes the addresses of the three neighbor blocks of a given 8x8 block (current block),
- the DC inverse prediction: an actor that computes a gradient between the already calculated neighbor blocks DCf coefficients, This gradient is used for the reconstruction of the DC coefficient of the current block. This actor is associated with the addressing actor to form the DC reconstruction unit,

- the AC inverse scan: depending on the value of the AC prediction calculated in the DC reconstruction unit, this actor inverts the one dimensional array of coefficients in zigzag or alternate the vertical or horizontal scan to 2D raster order,
- the inverse AC prediction: some specific AC coefficients already flagged in the bitstream are decoded in this actor,
- the inverse quantization: an algorithm realizing the inverse of the quantization table of the encoder,
- the inverse DCT2D: also an inverse algorithm of the DCT applied in the encoder.

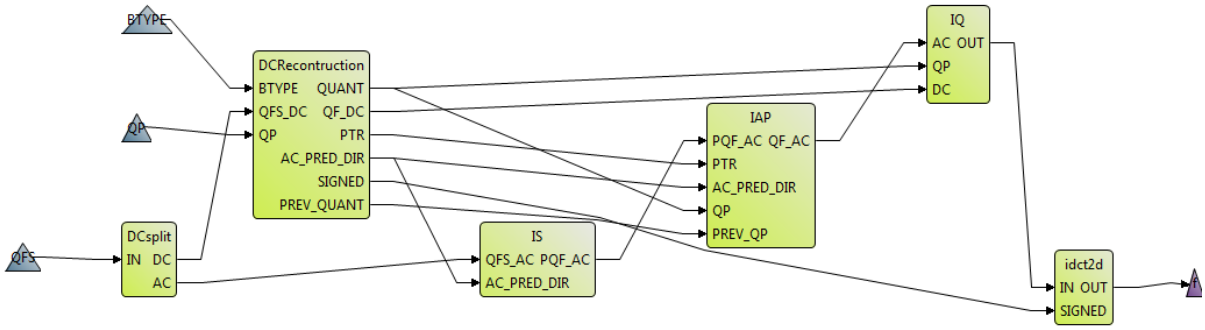

Figure 6.5: MPEG-4 part 2 texture decoder with RVC actors

► **The motion decoder** So called the inter decoder, this unit reconstructs the blocks of the frames using the motion vectors. This is the inverse algorithm of motion estimation and compensation used in the encoder. As presented in Figure 6.6, it consists of:

- a management address actor that generates the frame buffer addresses for motion compensation,
- a framebuffer actor: this actor stores the frames necessary for the motion compensation, It reads the "Btype" information in the input, This information indicates to the actor whether to read a new VOP or to read a motion vector or

to store a block. This actor is the most memory consuming of the MPEG-4 part 2 design,

- an interpolation actor,
- an addition actor that merges the texture and the motion decompressed data to form the final pixel values.

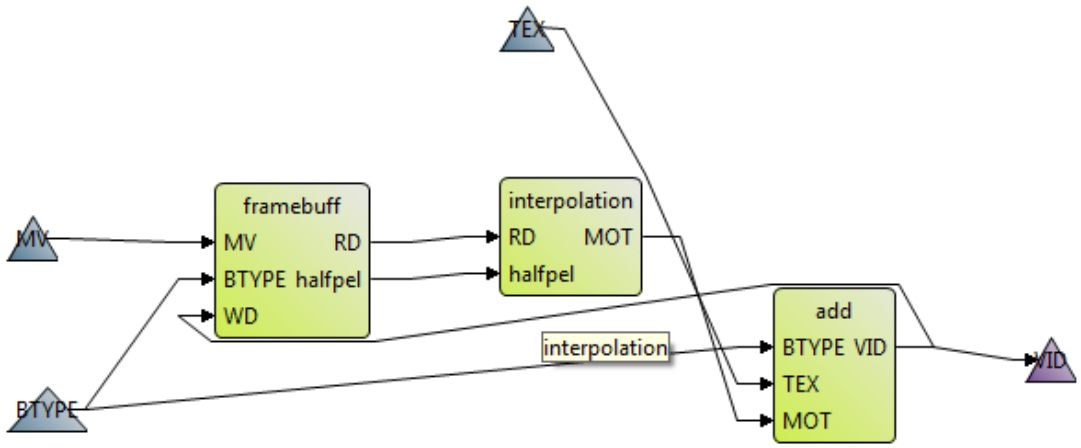

Figure 6.6: MPEG-4 part 2 motion decoder with RVC actors

► **The merger** This is the final step of the decoding process. The merger reads the separated Y, U an V decoded blocks (16x16 block size of Y, 8x8 block size of U and 8x8 block size of V) and merges them into a unique image signal YUV that can be displayed as the original video.

The MPEG-4 part 2 Simple Profile has been developed with different architectures and different abstraction levels, all realizing the same algorithm.

6.1.3 Serialized architecture

This architecture is very similar the Ericsson one but the algorithms are changed so that they do not contain the parallelism between the luminance and the chrominances decoding blocks. The texture and motion decoding actors are programmed

to process successive streams of four blocks Y, one block Cr and one block Cb. This design reduces considerably the area consumption of the design implementation.

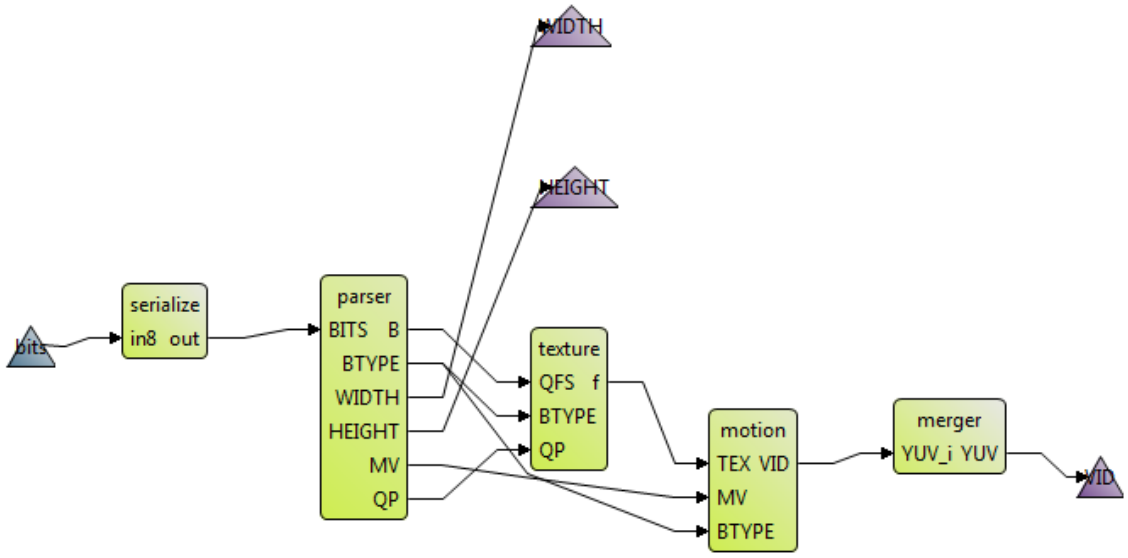

Figure 6.7: Serialized architecture of MPEG-4 part 2 decoder

6.2 MPEG-4 part 10 Profiles

The MPEG-4 part 10, so called MPEG-AVC, was established in 2003. Since, several profiles have been developed to deal with different coding contexts. Figure 6.8 illustrates the different functionalities and techniques existing in the standard and the way they are shared between profiles. AVC contains two entropy encoding modes:

- the Context-adaptive variable-length coding (CAVLC) used by all profiles,
- the Context-adaptive binary arithmetic coding (CABAC) used by the PHB, the FREXT and the main profile.

The most reduced profile is the CBP. The main profile includes the CBP functionalities added to the "weighted prediction" that computes the motion vectors and the "B slice" which is a technique that searches the predicted block in previous (backward) and future (forward) frames. The PHP extends the main profile with the "Intra 8x8" coding and the "customized quantization scaling matrices". The FREXT

extends the PHP with the "interlaced" coding techniques. Mainly, we notice four interlacing types:

- the full interlaced when all frames get interlaced,
- the paff which is a mix of progressive and interlaced frames,
- the mbaff a mix of interlaced and progressive macro blocks in each frame,
- a mix of interlaced and mbaff.

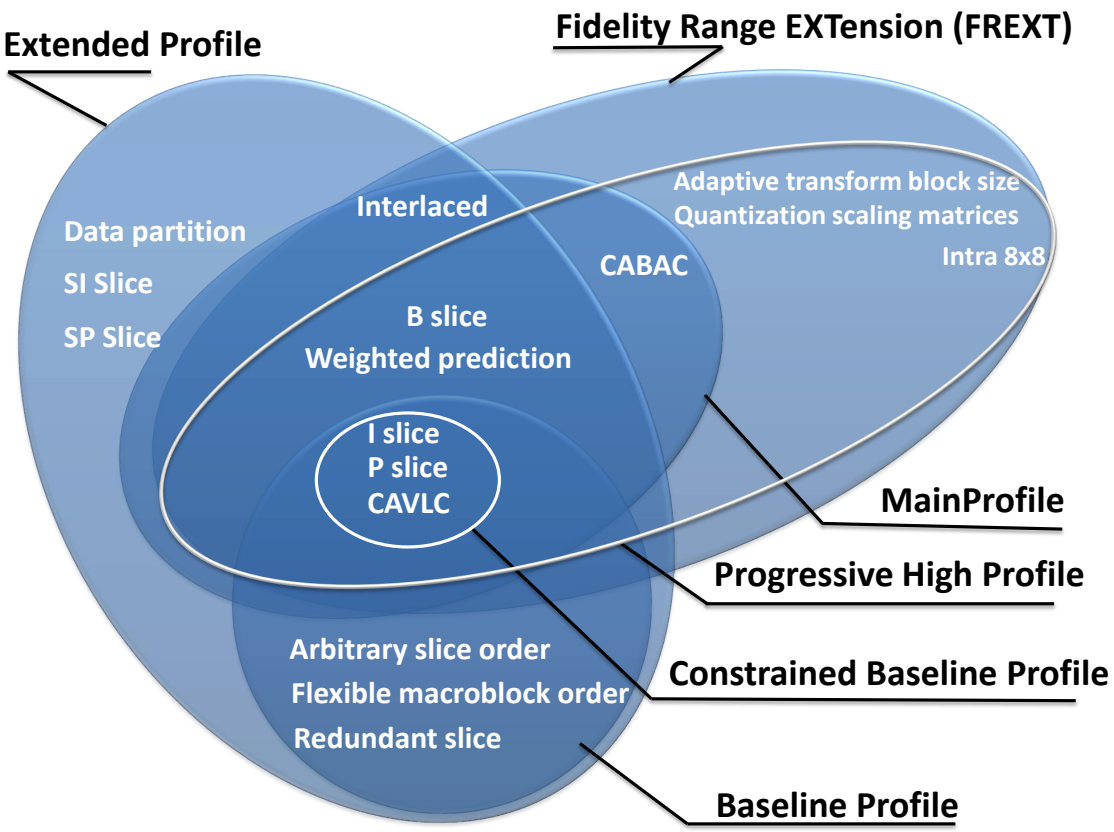

Figure 6.8: MPEG-4 part 10 profiles

In the VTL of MPEG-C part 4, three profiles are in development: the CBP (Constrained Baseline Profile), the Frext (Fidelity Range EXTension) and the PHP (Progressive High Profile) which is similar to the FREXT but it does not support field coding features. All MPEG-AVC decoders are very complex as presented in the

	Actors	Levels	Parser size kSLOC	Decoder size kSLOC
MPEG-4 SP	27	3	9.6	2.9
MPEG-4 AVC	45	6	19.8	3.9

Table 6.3: Composition of MPEG-4 Simple Profile and MPEG-4 Advanced Video Coding RVC-CAL description

comparison table with MPEG-4 part 2 SP 6.3.

Consequently, we present in the following the relatively simplest profile: the CBP.

The CBP is very similar to MPEG-4 part 2 Simple Profile. It is designed with a maximum of parallelism such as the Ericsson architecture presented above. As presented in Figure 6.9, this profile encapsulates three main parts:

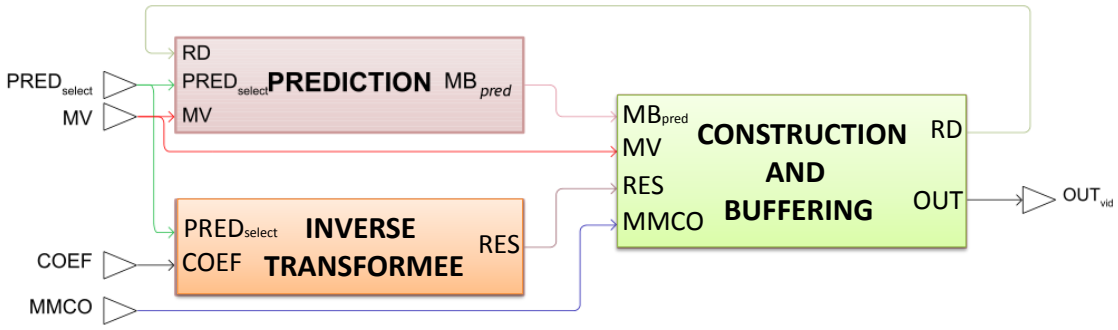

Figure 6.9: The CBP main decoding blocks

► **The prediction** For the prediction, the CBP insures the intra (spatial) and the inter (temporal) prediction modes. In figure 6.10, we can see that the designed architecture promotes a maximum of parallelism by launching the Intra 4x4, Intra 16x16 and the inter predictions simultaneously and using a selection actor to extract only the useful data depending on the value of the "MBType" signal. The RD signal contains already decoded data that is useful to reconstitute 4x4 or 16x16 blocks necessary for the extrapolation algorithms and the prediction. For the Inter prediction, the RD signal contains information from other reference frames. To get the prediction mode of the FREXT, it is just necessary to add the prediction Intra 8x8. Such operation is very easy using the graphs of MPEG-C part 4 designs.

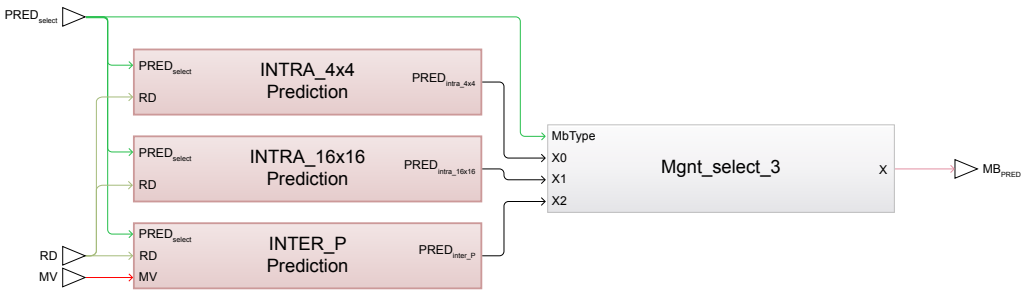

Figure 6.10: Design of the prediction block of the CBP profile

► **The inverse transformation** This part of the decoder is responsible of the reconstruction of the DC coefficients followed by the inverse quantization as presented in Figure 6.11.

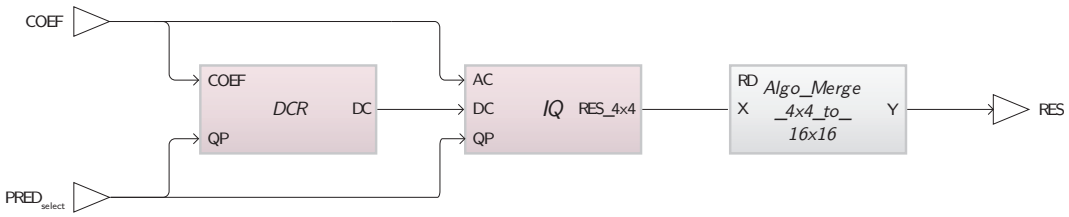

Figure 6.11: The texture decoder network of MPEG-AVC

This inverse transformation is unique for all AVC profiles since it is applied only on 4x4 blocks. The DCR block is the networks that reads the residual of the encoded data and produces the reconstructed DC coefficients in the output. The IQ block applies the inverse quantization and then a merger reconstructs the 16x16 blocks from the stream of 4x4 blocks.

► **The buffering and image reconstruction** Figure 6.12 illustrates the internal architecture of the construction block. The "add" actor is responsible of the addition of the residual with the the predicted coefficients to restitute the original image block. These blocks undergo a filtering process to delete the block effect that always appears in such block based coding processes. The last element of the network is the buffer block that stores the image for a later use during the Inter prediction.

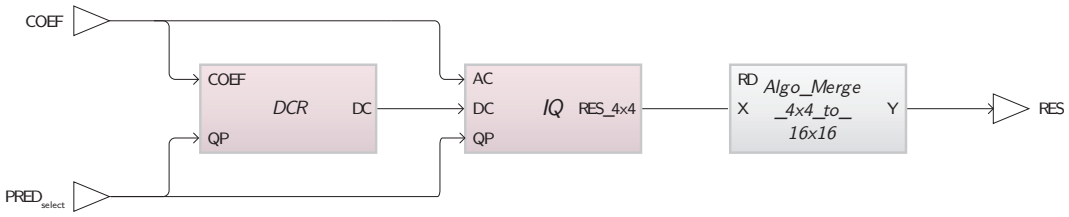

Figure 6.12: Architecture of the construction and buffering block in the CBP

6.3 Implementation and results

6.3.1 Functional validation

To remain faithful to our approaches, we started testing the transformations in the software platform. The first step was to validate the transformed mono-token code of some RVC-CAL designs and the second step was to generate the hardware. The tested designs are summarized in Table 6.4.

	SW verification	HW verification
MPEG-4 part 2 SP (Ericsson design)	✓	✓
MPEG-4 part 2 SP (MVG design)	✓	✓
MPEG-4 part 10 FREXT	✓	x
LAR	✓	✓
JPEG	✓	x
ACC-JPEG [42]	✓	x

Table 6.4: RVC-CAL designs tested in software and hardware platforms

All the designs that have been tested were transformed correctly to low level as checked in the software platform. The very high complexity of MPEG-4 part 10 FREXT profile proves the robustness of this work. The hardware implementation is restricted to MPEG-4 part 2 SP and the LAR and it is explained by the fact that, for both designs, we have a reference hardware oriented design. These references are very important to make comparisons and assess our methodologies.

6.3.2 Hardware implementation

We used the automatic transformations presented in Chapter 5 in Orcc to transform the multi-token firing rules of the MPEG-4 part 2 Ericsson design explained above. In our first tests, we omitted the inter decoder part because it is very memory consuming and the hardware generator is unable to manage it. The HDL generated

designs of the automatically transformed code and the optimized code were implemented on a Virtex4 (xc4vlx160-12ff1148) FPGA. This FPGA encapsulates several logical architectures such as *Slices*, LUTs and DSP blocks. The Slice is composed of two look up tables (LUT) and a flip-flop which is a sequential logical circuit that has two stable states used to store state information. The Virtex 4 contains also logical circuits dedicated for input and output ports (IOB) and also BRAM memory blocks. The obtained area consumption results presented in Table 6.5. The removal of read actions buffers and process actions had an important impact on the area consumption since it has decreased about 50%.

Criterion	Transformed design	Optimized design
Slice Flip Flops	21,624/135,168 (15%)	13,575/135,168 (10%)
Occupied Slices	45,574/67,584 (67%)	18,178/67,584 (26%)
4 input LUTs	68,962/135,168 (51%)	34,333/135,168 (25%)
FIFO16/BRAM16s	14/288 (4%)	14/288 (4%)
Bonded IOBs	107/768 (13%)	107/768 (13%)

Table 6.5: MPEG4 decoder area consumption

After the synthesis of the design, we applied a simulation stream of compressed videos. Table **??** below presents the timing results of a CIF (352x288) image size video.

Criterion	Transformed design	Optimized design
Maximum frequency (MHz)	26.4	26.67
Latency (μs)	381,8	306,4
Cadency (MHz)	1,9	2.33
Processing time (ms/image)	13,55	11,01
Throughput frequency (MHz)	1,8	2,2
Global image processing (FPS)	73,8	90,82

Table 6.6: MPEG4 decoder timing results

We notice that timing results were partially improved. This is due to the presence of division operations in some actors. In our transformation we replaced divisions by an Euclidean division which is very costly and time consuming. The impact is noticeable since these divisions reduced the maximum frequency by 60%. Therefore, we applied the transformation on the inverse discrete cosine 2D transform (IDCT2D). We chose this actor because it contains very complex algorithm, functions and procedures. We tried to compare with an optimal low level architecture designed by Xilinx experts and also with an existing implementation study of a direct VHDL written algorithm in [56]. For a significant comparison, we used the

same implementation target of the study which is the Xilinx Spartan 3 XC3S4000. Timing and area consumption results comparison are presented in tables 6.7 and 6.8.

Criterion	Xilinx design	Transformed design	Optimized design	VHDL design
Maximum frequency (MHz)	37	37	43	41
Latency (μs)	11.52	82.7	28.4	*
Cadency (MHz)	30	18.49	21.7	71
Processing time (μs/64Tokens)	1.99	3.4	2.8	0.89
Throughput frequency (MHz)	26.62	0.72	2.43	62.4
Global image processing (FPS)	1064	31	101	2518

Table 6.7: IDCT2D timing results

Criterion	Xilinx design	Transformed design	Optimized design	VHDL design
Slice Flip Flops	1415/55296(2%)	4002/55296(7%)	2113/55296(3%)	*
Occupied Slices	1308/27648(4%)	5238/27648(18%)	2523/27648(9%)	3571/27648(12%)
4 input LUTs	2260/55296(4%)	9861/55296 17%)	4777/55296(8%)	4640/55296(8%)
Bonded IOBs	48/489(9%)	49/489(10%)	49/489(10%)	*

Table 6.8: IDCT2D area consumption

Obviously, Table 6.8 reveals that area results for the optimized design are very close to those of the Xilinx low level design. This property is noted for all actors containing more computing algorithms than data control and management algorithms. Concerning the area consumption of the VHDL design, it is expectable to find results nearby the optimal design and clearly worse than the Xilinx design. This is due to the synthesis constraints indicated in [56] that favor treatment speed in spite of the surface. This is what explains also the very high FPS rate of the design presented in Table 6.7. Timing results of the other designs show that the optimized design performances are far from the optimal Xilinx design. This is due to the low level architecture made by Xilinx experts which is completely different and oriented for hardware generation. This architecture is a pipelined set of actors realizing the IDCT2D (rowsort, fairmerge, IDCT1D, seperate, transpose, retranspose and clip) which is a relatively complex design compared with the high level IDCT2D code used for the transformation.

After comparing with the Xilinx design and a VHDL directly written design, we compared our results with existing generation tools and we considered GAUT hardware generator. This tool is an academic high level synthesizer from C to VHDL. It

* Not mentioned in the literature

extracts the parallelism and creates a scheduled dependency graph made of elementary operators. Potentially, GAUT synthesizes a pipelined design with memory unit, communication interface and a processing unit. However, such as most existing hardware generators, GAUT is not able to manage a system level design with very high complexity and a variety of processing algorithms. Moreover, there are so many restrictions on the C input code to have a functioning design. As it was impossible to test the whole MPEG 4 decoder, we restricted the choice to the IDCT2D algorithm to have a comparison with previously presented results.

The IDCT2D is so generated with GAUT and we obtained the results of table 6.9 below:

Criterion	GAUT design	Optimized design
Slice Flip Flops	2,080/135,168 (2%)	1,988/135,168 (2%)
Occupied Slices	2,477/67,584 (3%)	2,353/67,584 (3%)
4 input LUTs	4,243/135,168 (3%)	4,458/135,168 (3%)
Bonded IOBs	627/768 (81%)	49/768 (6%)

Table 6.9: IDCT2D area consumption with GAUT

Results show that the optimized transformation generates a better design even for the specific case of study of the IDCT2D. The important logic consumed by the IOBs implementation generated with GAUT can be explained by the fact that the synthesizer of GAUT generates a port for each input data. In the case of the IDCT2D, GAUT's implementation includes 64 input ports and 64 output ports.

6.4 Conclusion

This chapter presented the main decoding techniques and profiles of MPEG-4. It also presented an overview of the existing decoders architectures of MPEG-4 part 2 (Ericsson, Xilinx and MVG). These decoders are already developed with RVC-CAL in the VTL of the RVC standard. The Ericsson design is considered as a complete algorithm example to test the performance of the methodology and the transformation, explained in chapters 4 and 5. The generated and the optimized hardware designs are implemented on Xilinx FPGA target and the results revealed to be very interesting compared to the hardware reference design of Xilinx and also to existing academic tools such as GAUT.

This test of the automatic transformation concerns a video decoder. In the next chapter, the transformation will be applied on a still image coder called LAR.

Chapter 7

Still image case of study: the LAR codec

7.1 LAR principle

The LAR (Local Adaptive Resolution) coder is developed at the IETR/ INSA of Rennes laboratory. It is based on the idea that the coding process can be adapted to the local activity of the image. In the still image classic coding methods like JPEG, an image is divided into a set of fixed size macro blocks (8x8 or 16x16) and all the coding algorithms are applied similarly on these blocks. Consequently, the image coding process is going to be applied the same way on regions containing an important activity (black block of Figure 7.1) and regions containing a very low variation (white block of Figure 7.1). Thus, it is possible to lose some details of the image especially in the contours where the correlation decreases.

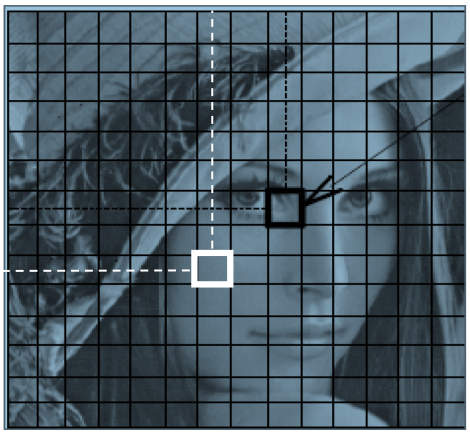

Figure 7.1: Monotonic blocks decomposition

The LAR coder uses a morphological gradient, explained later, to detect the uniformity or the variation of the local luminance. This gradient outputs a map of variable size blocks such as the higher the activity the lower the size [23].

Another aspect of the LAR coding is based on the consideration that an image is a superposition of a global information image (mean blocks image), and the local texture image, which is given by the difference between the original image and the global one. This principle is modeled by:

$$I = \overline{I} + (\underbrace{I - \overline{I}}_{E})$$

Where I is the original image, $\overline{I}$ is the global information image and $I - \overline{I}$ is the error image E. The dynamic range of the error image is consequently dependent on the local activity. In uniform regions, $\overline{I}$ values are close or equal to I consequently $I - \overline{I}$ values are around zero with a low dynamic range.

Considering these principles, the LAR coder concept (Figure 7.2) is composed of two parts: the FLAT LAR [21] which is the part insuring the global information

coding and the spectral part which is the error spectral coder.

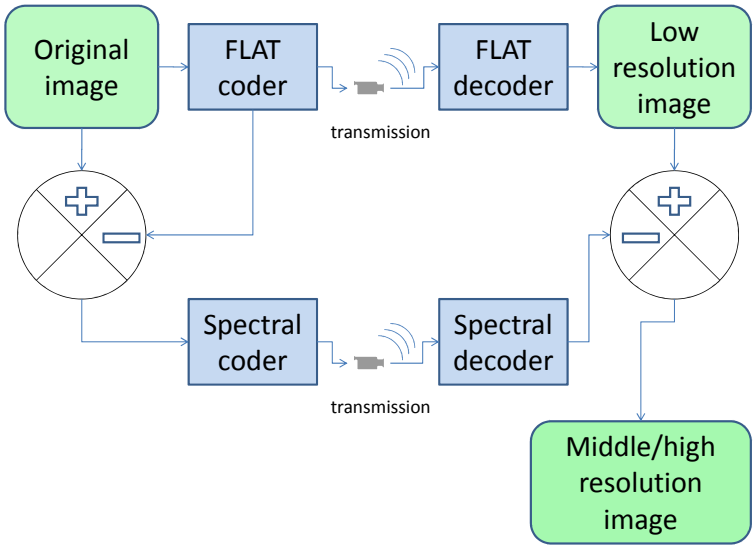

Figure 7.2: LAR baseline concept

Different profiles have been designed to fit with different types of application as presented in Figure 7.3. After the baseline, a pyramidal profile was added to encapsulate the context classification, the rate control etc. The extended profile adds the scalability, the region of interest, the cryptography etc. In this thesis, we focused on the baseline coder. Its mechanisms are detailed in the following.

7.1.1 FLAT LAR

As shown in Figure 7.4, the Flat LAR is the spatial coder of the LAR. It is composed of three main parts: the partitioning, the block mean value computation and the DPCM (Differential Pulse Coding Modulation).

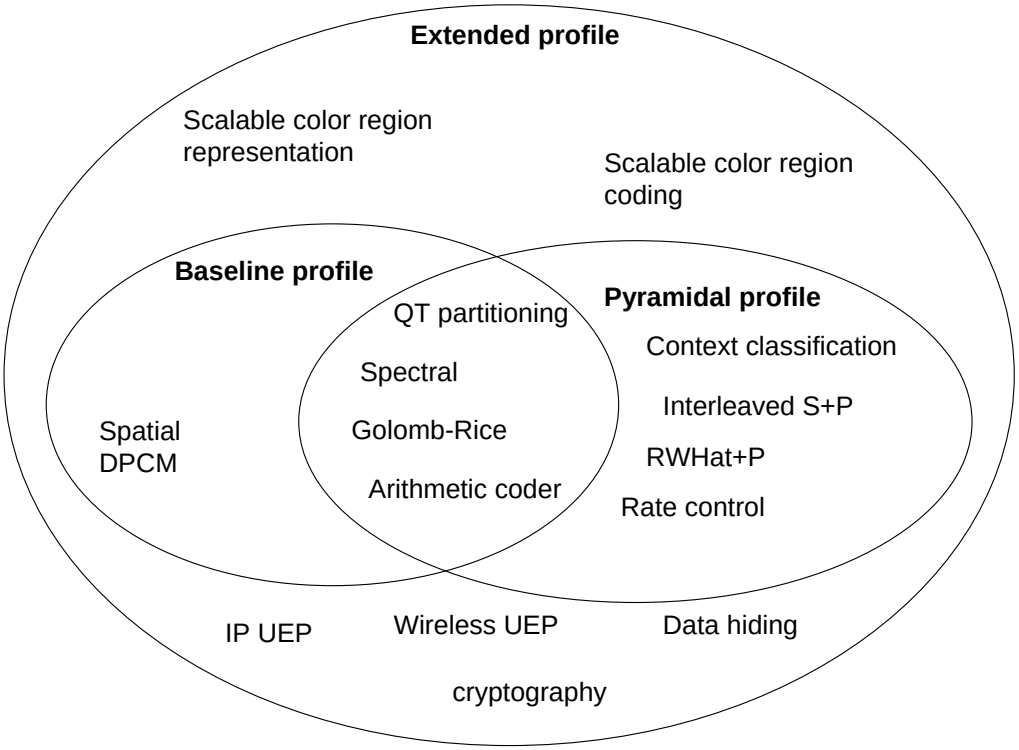

Figure 7.3: LAR profiles

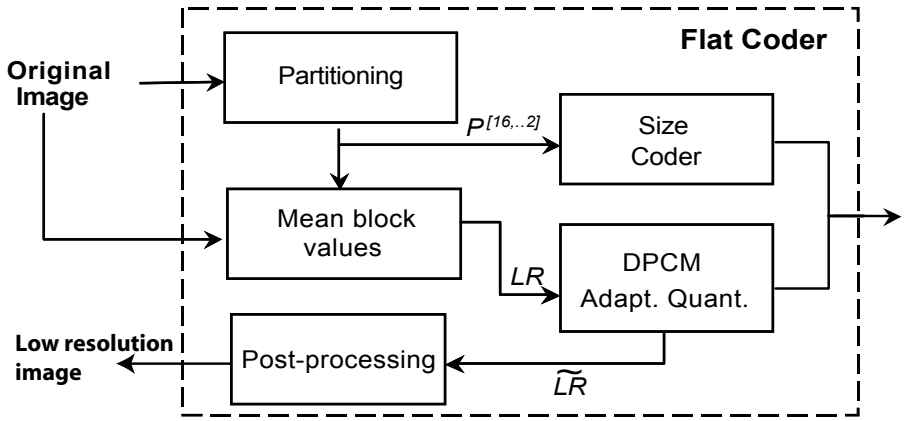

Figure 7.4: FLAT LAR architecture

7.1.1.1 Partitioning

To have a variable block size representation, it is necessary to adopt a partition topology. The LAR considers the Quadtree partition. A Quadtree partitioning is defined with $P^{[Nmax..Nmin]}$ where Nmax and Nmin are the maximum and minimum of per-

mitted block size. These size are equal to a power of 2. The Quadtree starts by clipping the image into blocks of $NmaxXNmax$ size. It applies a homogeneity test [70, 73] on these blocks. If the homogeneity is not met then the block is divided into son blocks. Otherwise, it stops the division and considers the current size block. The Quadtree process of the LAR is based on a morphological gradient that checks if the activity is important in that block of the image. The criterion test consists of comparing the difference between the maximum (MAX) and the minimum (MIN) luminance values of the block with a threshold THD as explained in the algorithm below:

```
If (MAX - MIN) > THD then
  size = actual_size;
else
  size = actual_size / 2;
  -- the considered block size is now: actual_size / 2 squared
end
```

This process is recursively applied on the whole image blocks and the output of the overall is the block size image presented in Figure 7.5. The block size image is necessary for the remaining coding processes.

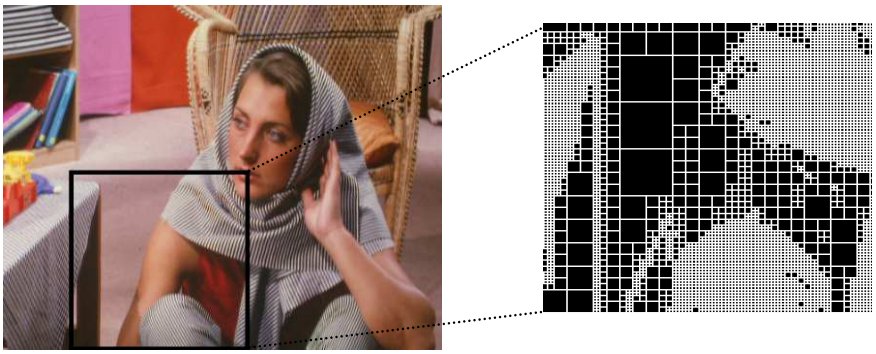

Figure 7.5: Block size image example

7.1.1.2 Block mean values computation process

This process generates the low resolution image $(LR_y : LR_Cr : LR_Cb)$. Figure 7.6 shows an original image and the corresponding low resolution image. For each block, the mean value is computed and put in the block as presented in the example of Figure 7.7. For each pixel p with coordinates (x, y), the algorithm is defined as: $LR(x, y) = \frac{1}{N^2} \sum_{k=0}^{N-1} \sum_{m=0}^{N-1} I([\frac{x}{N}] \times N + k, [\frac{y}{N}] \times N + m)$

such as $N = size(x, y)$, the size of the block containing the pixel.

Figure 7.6: Low resolution image example

7.1.1.3 The DPCM

The Quadtree partitioning and the block mean value involve a non-uniform sub-sampling of the image which has an important impact not only on the compression rate but also on the global bit rate that decreases during the prediction and quantization presented in Figure 7.8.

► **Prediction of luminance block** The observation that a pixel value is mostly equal to a neighbor one led to the following estimation algorithm. If we consider the pixels in Figure 7.9, X value is estimated with the algorithm:

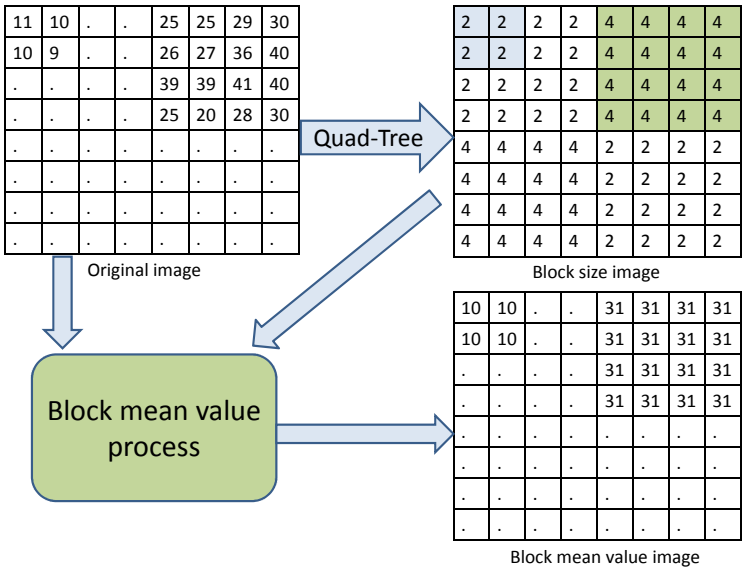

Figure 7.7: Block mean value process example

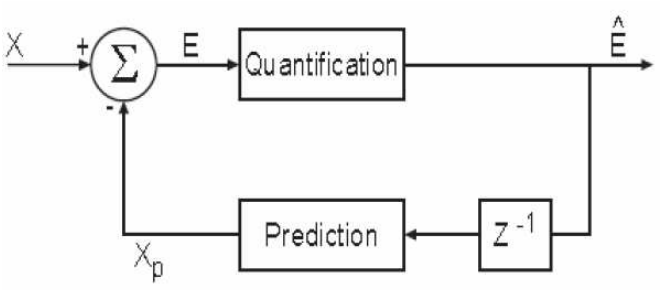

Figure 7.8: The DPCM principle

$$X = \begin{cases} A & If|B-C| < |A-B| \\ C & else \end{cases}$$

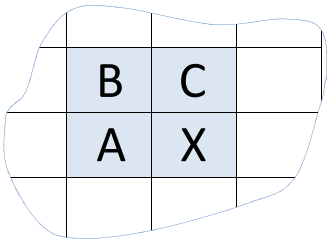

Figure 7.9: DPCM prediction of neighbor pixels

► **Quantization** Experiments on the eye perception proved that we are less sensitive to the variation of luminance and color in contours zones also called high visual frequency areas [76, 54]. Other studies show that the block degradation during the linear quantization of a block is inversely proportional to its size. Based on these facts, the LAR applies quantization on the mean block values as presented in table 7.1.

Size	16	8	4	2
q_N	2	4	8	16

Table 7.1: Quantization according to the block size

Finally, the outputs of the FLAR are the quantized image and the error image explained above as the difference between the original image and the quantized one. Both images have low dynamics which is very important in the compression performances of the LAR. Some optimizations have been added for the design by introducing pipeline in the DPCM as presented in Figure 7.10.

In the following, we present the frequency (or spectral) coder which is the second compression level.

7.1.2 Spectral coder: The Hadamard transform

The spectral coder presented in Figure 7.11, also called the frequency coder, is composed of a variable block size Hadamard transform [62] and the Golomb-Rice [30]

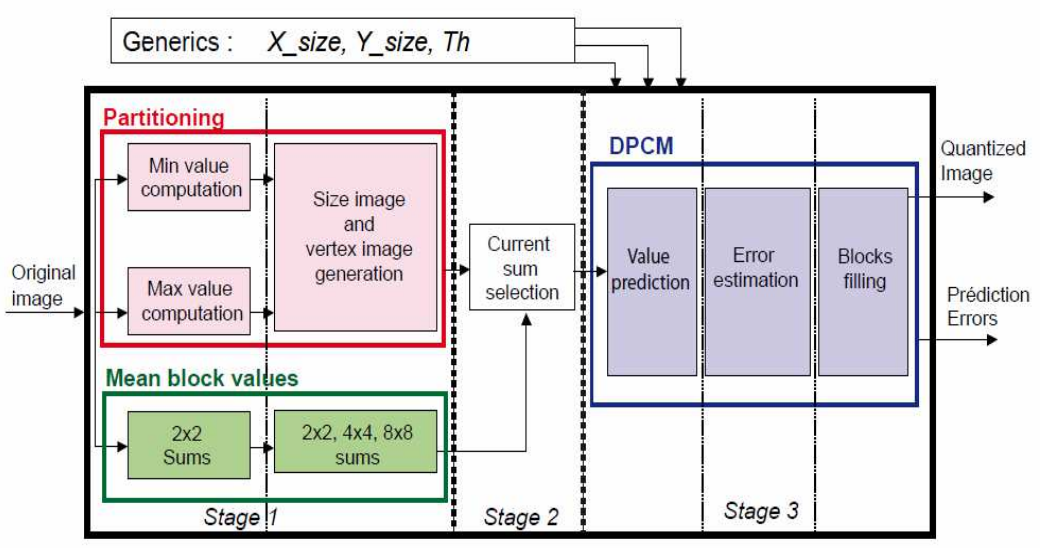

Figure 7.10: DPCM pipelined architecture

[64] entropy coder. The Hadamard transform derives from a generalized class of the Fourier transform. It consists in a multiplication of a 2^mx2^m matrix by an Hadamard matrix (H_m) that has the same size.

The transform is defined as follows:

H_0 is the identity matrix so $H_0 = 1$. For any m>0, H_m is then deducted recursively by:

$$H_m = (\frac{1}{\sqrt{2}})^m \left| \begin{array}{cc} H_{m-1} & H_{m-1} \\ H_{m-1} & \text{-}H_{m-1} \end{array} \right|$$

Here are examples of Hadamard matrices:

$H_0 = 1$,

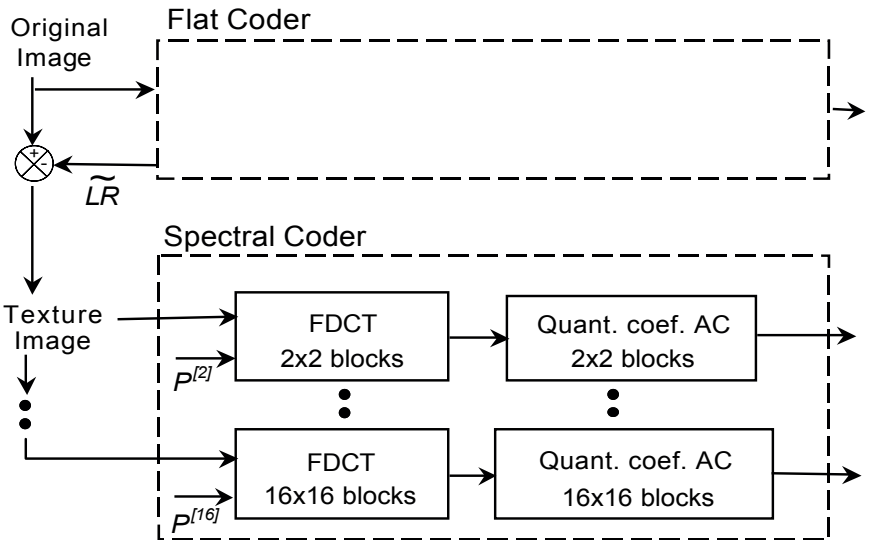

Figure 7.11: LAR spectral coder architecture

$$H_1 = \frac{1}{\sqrt{2}} \begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix},$$

$$H_2 = \frac{1}{2} \begin{vmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{vmatrix}, \text{ etc ...}$$

The Hadamard transform resulting operations are simple additions and subtractions. Therefore, it is very easy to be implemented in hardware circuits compared with the matrix multiplication of the Discrete Cosine Transform (DCT) of the JPEG or MPEG coders. Moreover, the Hadamard matrix is symmetric and reversible. This property is very important for the decoding step because it is possible to use the same encoding block for the decoding one. It is important to take in consideration the accumulation of the $\frac{1}{\sqrt{2}}$. When an hadamard H_n is applied and n is an odd number , the $\frac{1}{\sqrt{2}}$ can cause many problems of computations. Consequently, the proposed solution of the LAR consists in realizing a double multiplication by the Hadamard matrix and its transpose, such as: for each matrix M_n
$Hadamard(M) = transpose(M * H)) * H$
and the divider is always a power of 2 and the problem can be solved using simple right shifts and this division is called a *normalization*.

7.1.3 Entropic coder: The Golomb Rice bitstream

The Golomb Rice coder is a bitstream generator that uses a tunable parameter $k > 0$ as a divisor and thus transforms an input value into a quotient q and a remainder r. The quotient is transformed using a unary coding and the remainder is encoded with truncated binary encoding, with a zero bit separating the two parts. The length of the quotient cannot be fix but the remainder length never exceeds the $log_2(k)$.
If $k = 1$ then ($input\ div\ k = input$), the Golomb Rice is equivalent to a unary coder.

Let us consider a coding using a parameter $k = 4$. For an input value $I = 15$, we have $q = 15\ div\ 4 = 3$ and so coded into 111 or 1110 with the separation zero bit. For the remainder, $r = 3$ and coded into 11. I is consequently transformed into 111011.

It is clear that the parameter k is crucial for the length of the bitstream because the increase of k involves a quotient decrease but also a remainder increase. This dilemma can be satisfied by choosing the best k value. Nevertheless, this choice depends on the tokens values. So the LAR proposes a visitor that peeks a sample of tokens (one token per 10 for example), computes the length of their encoding output using a set of k parameters and keeps the best parameter that allows the shortest output.

7.2 Achieved architectures

The LAR coding is dependent from the content of the image. It applies in the Quad-Tree a morphological gradient to extract information about the local activity on the image. The output is the block size image represented by variable size blocks: 2x2, 4x4 or 8x8. Using the block size image, the Hadamard transform applies the adequate transform on the corresponding block [45, 46]. It means that if we have a block size of 2x2 in the size image this block will undergo a 2x2 Hadamard (H_1) and a normalization specific to the 2x2 blocks. This process is identically applied for 4x4 and 8x8 blocks. A quantization step, adapted to current block size, is applied on the Hadamard output image. For each block size, a quantization matrix is predefined. Practically, the normalization during the Hadamard transform is postponed to be achieved with the quantization step so that to decrease the noise due to successive divisions.

The implemented LAR is presented in Figure 7.12. As a first step, the memory management block stores the pixels values of the original image line by line. Once an 8x8 block is obtained, the actor divides it into sixteen 2x2 blocks and sends them in a specific order as presented in Figure 7.13. This order is very important to improve the performance of remaining actors. In fact, considering the Figure 7.13, when the tokens are so ordered the first 4 tokens correspond to the first 2x2 block, the first 16 tokens to the first 4x4 block etc ... Consequently, and as presented in

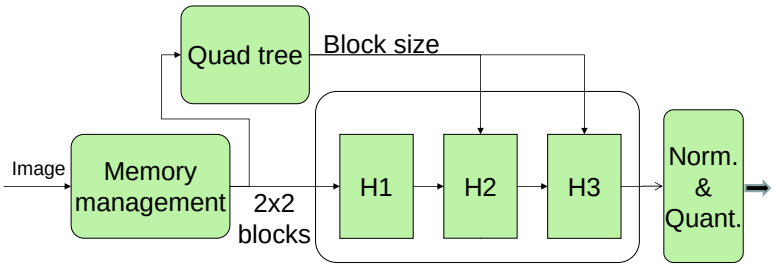

Figure 7.12: LAR baseline developed model

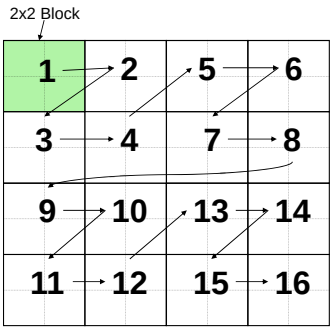

Figure 7.13: Memory management unit output order

Figure 7.15, the output of the H1 is automatically the input of the H2 and the output of the H2 is automatically the input of the H3.

In the Quad-Tree, this order is also crucial. As presented in Figure 7.14, the superposition of the same actor (max for example) three times provides in the output of the first actor the maximum values of 2x2 blocks, in the output of the second actor the maximum values of 4x4 block and finally the maximum values of 8x8 blocks in

the output of the third one. Using the maximum values and the minimum ones, the morphological gradient in the Gradstep actors can process to extract the block size image [45, 44]. The same tip is used to calculate the block sums with three superposed sum actors. The block mean value actor considers the sums and the sizes to build the block mean value image.

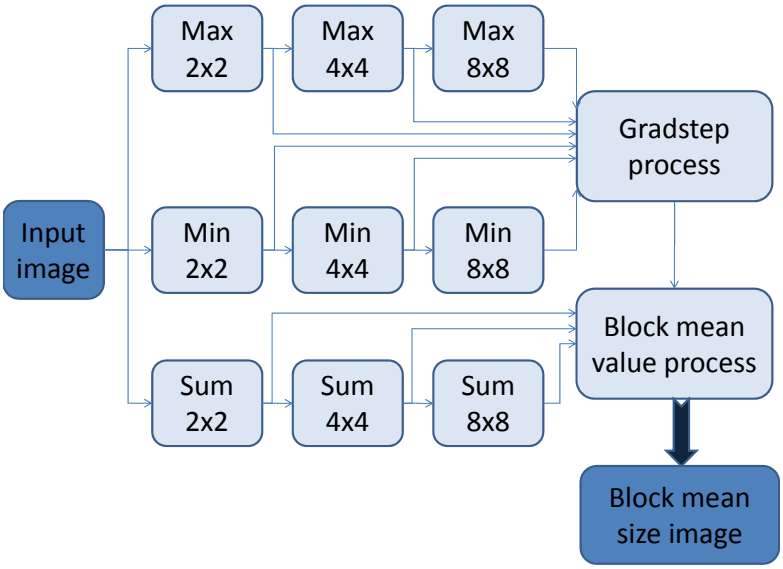

Figure 7.14: Quad-Tree design

In the spectral coder, this scan order is decisive. Indeed, let us consider an (H_1) transform applied on a 2x2 size matrix: $M_2 = \begin{vmatrix} x1 & x2 \\ x3 & x4 \end{vmatrix}$.

The application of the equation (1) outputs the matrix:

$$M_2' = \begin{vmatrix} x1+x2+x3+x4 & x1-x2+x3-x4 \\ x1+x2-x3-x4 & x1-x2-x3+x4 \end{vmatrix}.$$

If we apply the same equation now on a 4x4 size matrix:

$$M_4 = \begin{vmatrix} x1 & x2 & x3 & x4 \\ x5 & x6 & x7 & x8 \\ x9 & x10 & x11 & x12 \\ x13 & x14 & x15 & x16 \end{vmatrix}$$

we obtain the matrix: $M'_4 = \begin{vmatrix} h1 & h2 & h3 & h4 \\ h5 & h6 & h7 & h8 \\ h9 & h10 & h11 & h12 \\ h13 & h14 & h15 & h16 \end{vmatrix}$

such as:

$h1 = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + x11 + x12 + x13 + x14 + x15 + x16$
$h2 = x1 + x2 + x3 + x4 - x5 - x6 - x7 - x8 + x9 + x10 + x11 + x12 - x13 - x14 - x15 - x16$
...

which can be seen also like:

$h1 = X1 + X2 + X3 + X4$
$h2 = X1 - X2 + X3 - X4$
...

where

$X1 = x1 + x2 + x3 + x4$
$X2 = x5 + x6 + x7 + x8$
$X3 = x9 + x10 + x11 + x12$
$X4 = x13 + x14 + x15 + x16$

It is obvious that the H_2 is a double reproduction of the H_1 with a certain data management and so (H_2) transform can be achieved using the (H_1) results of the four 2x2 blocks constituting the 4x4 block. The same observation can be made for the (H_3) one. This ascertainment is very important to decrease the complexity of the process. In fact, the Hadamard transform of the LAR applies an (H_1) transform for the whole image then it applies the (H_2) transform only for the 4x4 and 8x8 blocks and the (H_3) transform only for the 8x8 blocks. The (H_2) and the (H_3) transforms are different from the full transforms as they are much less complex. Consequently, as shown in Figure 7.15, we designed the H2 and the H3 using H1 actors associated with memory management units. They sort tokens in the adequate order and, considering the block size, whether the block is going to undergo the transform or not. It is very important to mention that almost actors have been developed with generic variables for memory sizes or gradsteps which means that the design are flexible for easy transformation from an image size to another or for adding higher Hadamard sizes (H4, H5 ...).

Moreover, other architectures have been designed for the Hadamard transform:

▶ **The multi-port design** The token per token data transfer is one of the most time consuming features of the developed design. A first idea to reduce this time

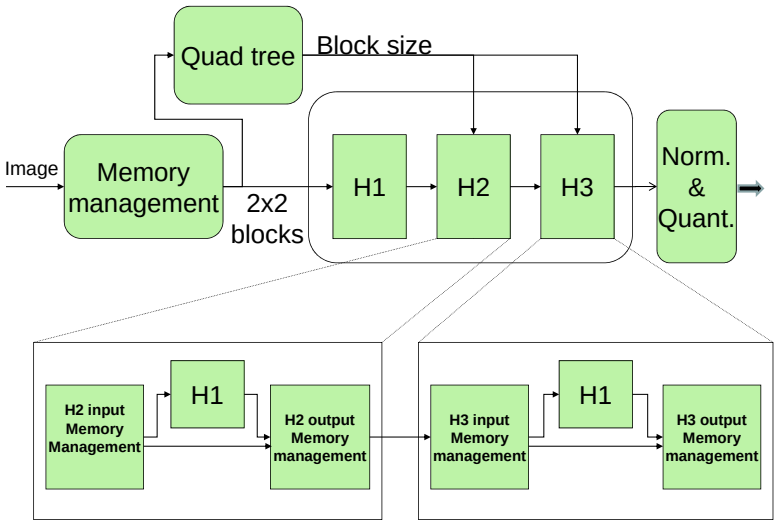

Figure 7.15: Hadamard hardware architecture

consumption was to parallelize the transfer using a multi-port design (Figure 7.16). It is like we unroll the repeat structures into a set of ports. However, it was necessary to add more complexity in the algorithms to make the actors write correctly in the adequate ports and thus we lose the advantage of using the data management. Besides, such a design is very consuming in terms of memory since it exceeds the available number of FIFOs in some FPGA.

► **The concatenation design** To reduce the number of FIFOs, we proposed to concatenate the data in a larger FIFO size (Figure 7.17). The principle is to replace the repeat into a bus containing a set of concatenated data. In spite of producing token per token, the data management concatenates every 4 tokens into a 4x*data_size* bus. The H1 block concatenates every 16 tokens into a 16x*data_size* bus etc . The drawback of such architecture is the fact it is necessary to add the concatenation and the deconcatenation algorithms in the input and the output of each actor which is also time consuming compared with a token per token data transfer. Another problem appeared later concerning the hardware synthesizers that are not able to compile a very large FIFO. The ISE tool of Xilinx, for example, cannot exceed a threshold of 32 bits.

► **The full Hadamard design** The parallelism of the design was a subject of study. Indeed, the use of the results of an H_k to rapidly make an H_{k+1} involves a sequential design since we create a data dependence. We designed the architec-

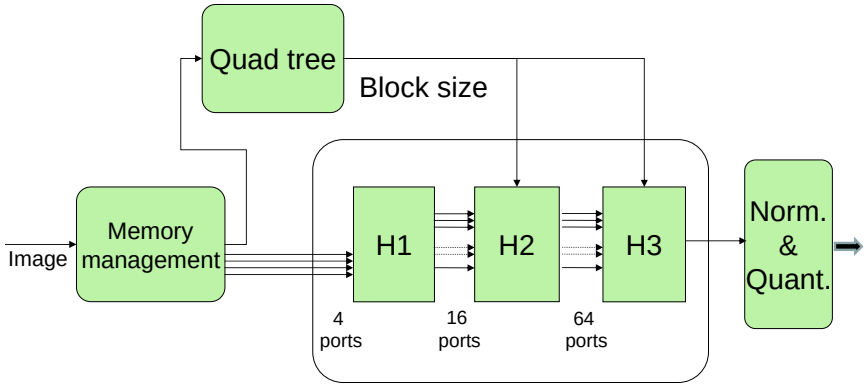

Figure 7.16: Hadamard multi-port architecture

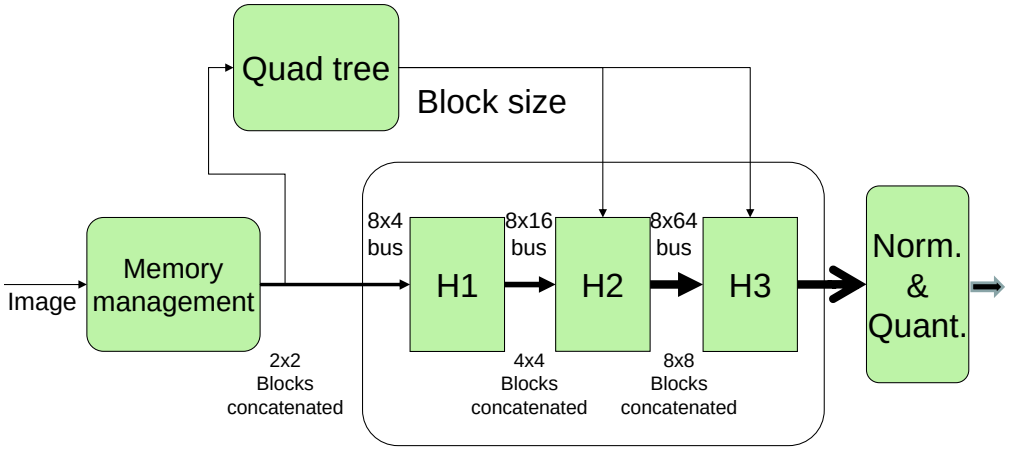

Figure 7.17: Hadamard concatenation architecture

ture of Figure 7.18 where every H_k block is a full treatment block. A sorting actor was added to collect data from all Hadamard actors and puts the correct data in the correct place of the output image depending on the size image. Theoretically, this method is faster that a sequential one and logically more area consuming but simulations shown an important latency that decreases the performance compared with the expected one. This latency is, in fact, due to the type of memory created by the Xilinx synthesizer. These memories are BRAM with two read/write buses. So an H2 block for example is designed to read 16 tokens in a local buffer and process them with an algorithm that uses all of them in a set of instructions. Indeed, the hardware design requires at least 8 clock periods to read the 16 tokens from the BRAM to execute the algorithm. That is why the resulting design performance is so close to the sequential one.

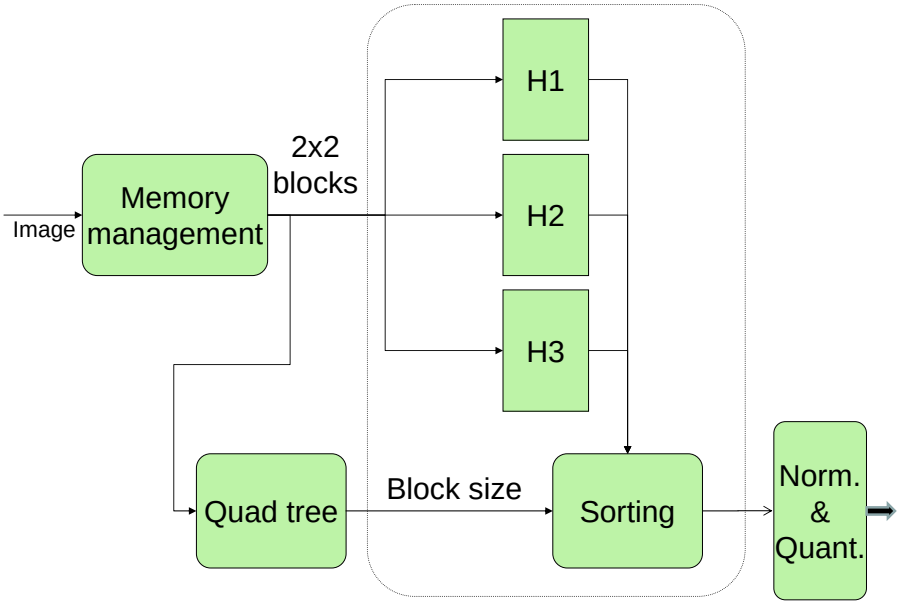

Figure 7.18: Hadamard parallel architecture

► **The sequential and no-conditional Hadamard design** In this design, we considered the same treatment blocks of Figure 7.15 but we eliminated the conditional structures of the Hadamard blocks and we added the sorting actor for that purpose. Consequently, Hadamard actors are going to behave the same way on every block of the input image which means that the H2 is applied on all 4x4 blocks, and the H3 is applied on the 8x8 blocks in an automatic way. Finally, the sorting ac-

tor collects all results and fills in the output image using the size image as presented in Figure 7.19 This architecture is very interesting in terms of global compromise between area and time performance.

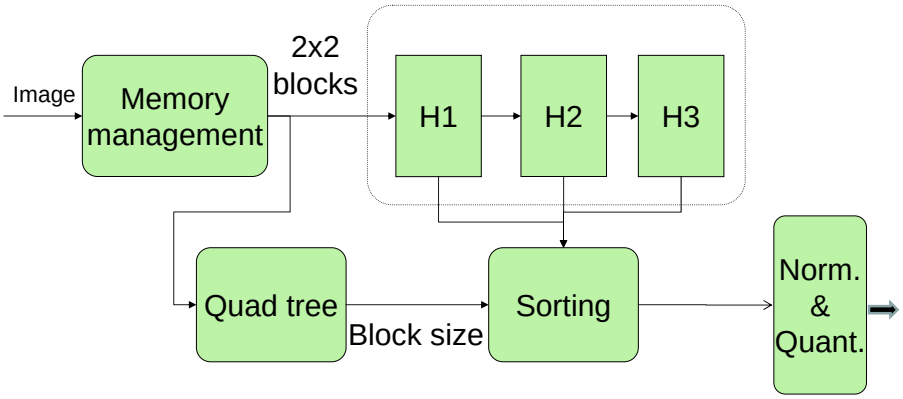

Figure 7.19: Hadamard sequential architecture with sorting actor

Among all the presented architectures, we adopted the one of Figure 7.15 because it is the easiest to develop and because it reuses many actors for different algorithms. These aspects match the fact of using the MPEG-RVC standard that recommends the easiest development solution, the highest level and the reusability of actors. For the same reasons, we neglected an optimized design using the ping-pong pipeline algorithm explained in Section 4.3. The considered architecture is the subject of the implementation study of the next Section.

7.3 Design implementation

The architecture presented above is developed with RVC-CAL using the methodology of Chapter 4 where a software platform is used for the validation of the high and low level designs. The transition from high level to low level is done using both of manual transformations and automatic transformation presented in Chapter 5. The

hardware generation process was applied on the 23 actors of the LAR using Orcc. The HDL generated code was implemented on a virtex4 (xc4vlx160-12ff1148). The area consumption results obtained are presented with those of manual transformation in Table 7.2.

Transformation	Automatic	Manual
Slice Flip Flops	20,452/135,168 (15%)	12,157/135,168 (8%)
Occupied Slices	47,576/67,584 (70%)	43,602/67,584 (67%)
4 input LUTs	59,868/135,168 (44%)	53,417/135,168 (39%)
Bonded IOBs	41/768 (5%)	41/768 (5%)

Table 7.2: LAR coder area consumption

After the synthesis of the design, we applied a simulation stream of compressed videos. Table 7.3 below presents the timing results of a CIF (352x288) image size video.

Transformation	Automatic	Manual
Development time	30%	100%
Maximum frequency (MHz)	61,43	85,27
Latency (ms)	0,42	0,12
Throughput frequency (MHz)	3,5	5,6
Processing time (ms/image)	35	19
Global image processing (FPS)	34	53

Table 7.3: LAR timing results

Concerning the area consumption, the occupied Slices and LUTs are almost equal which is a very satisfying result. The Slice Flip Flop difference can be explained by the fact of adding untagged actions which increases the complexity of the global FSM behavior. Flip Flops can be also used for local registers as bits memories and so it is a source of increase since the automatic transformation adds several intermediate registers for the treatment. The reason is that the transformation applies a general modification whatever the complexity of the actor.

Concerning the timing results, the automatic and the manual transformed designs performances remain close and acceptable. The latency difference is explained by the fact that the untagged actions, as always given priority over the rest of actions, promote the data reading. It means that, as long as there is data in the FIFO, the untagged action fires even if there are enough data to fire the processing actions. Nevertheless, it is very important to note the development time reduction

using high level description followed by the automatic transformation which can decrease the time to market by over 70%.

To have another aspect of the assessment of the used methodology, we considered a manually developed VHDL design of the FLAT LAR introduced in [22]. In the following, we present in Table 7.4 the area and timing performances compared with those of an RVC-CAL design.

Source code	VHDL		RVC-CAL	
	64x64	352x288	64x64	352x288
Development time (man/month)	4		1	
Internal memory (byte)	684	3470	3168	13088
4inputs LUTs for logical operations	1166	2463	7423	7423
Frequency (MHZ)	45.8	33	8.5	8
Processing time (ms)	0.09	3.1	0.33	11.7
Latency time (μs)	18.6	75	26.6	27.8
Global image processing (FPS)	9200	314	2804	85

Table 7.4: FLAT LAR: VHDL VS CAL comparison

As expected, the design directly written in VHDL has clearly better performance. We also note that the VHDL code was developed using a dedicated and pipelined architecture different from the Dataflow philosophy used in the RVC-CAL design but realizing the same global behavior. Nevertheless, RVC-CAL results remain acceptable for real time display.

7.4 Conclusion

In this chapter we presented the main notions of the LAR coder as a new context of the RVC-CAL applications for still image coders. To satisfy the algorithm specifications of the LAR, many architectures were developed whether to add more parallelism or to reduce the complexity using smart data management or a profit from already processed data. The architecture that have been adopted for the rest of the implementation is the highest-level and the simplest one which is an important property of the MPEG-RVC standard. Staying faithful to the conception flow, the considered design was developed and validated in a software platform before applying transformations to low level for hardware generation. We achieved the same design of the baseline of the LAR coder using a manual transformation and the automatic transformation in the IR of Orcc explained in Chapter 5. The implementation results were very interesting since the difference between the automatically and the manually transformed designs was not important and exactly as

we expected. The most important conclusion is that we implemented correctly the first still image coder developed in the VTL of MPEG-RVC using our approach and transformations. Of course, further optimizations in the transformation process can reveal better results. Some pipeline algorithms can be added to make performance close to those of the VHDL design.

Chapter 8

Conclusion and perspectives

8.1 Summary

This thesis was motivated by the increasing demand on rapid prototyping solutions for embedded systems. Indeed, the growing complexity of applications related to digital signal processing involves a considerable conception time. To manage this complexity and reduce the time-to-market of implementations, the Electronic System Level Design methodology was introduced to design the very complex applications at a high level description so called system level. This description is insured using Models of Computation related to Domain Specific Languages and graphs. This thesis work adopted the Dataflow MoC and a description language called RVC-CAL. This high level language is associated with a compilation platform that generates hardware and software implementations. The problem is that the existing hardware generation infrastructure presents several limitations related to slow existing procedures of validation. Moreover, some high level features of RVC-CAL that are omnipresent in video decoders codes but are not compliant with hardware compilers. The work achieved during this thesis aims to solve the limitations of hardware generation from Dataflow programs.

This document starts with the state of the art of our research activities. This part starts with a presentation of the state of the art of the conception methods of digital signal processing circuits. Considering the limitations of the existing methods to manage complex applications, we introduced the Electronic System Level Design methodology that allows the automatic generation of software and hardware codes using high level descriptions. We demonstrate the main advantages and drawbacks of the hardware and the software platforms to conclude with the importance of mixed architectures. These codes have to be synthesized into Register Transfer Level by applying high level synthesis techniques. That is why we presented some HLS techniques and tools.

After the state of the art, Chapter 3 presents the Reconfigurable Video Coding standard and explains the motivations behind this standard to improve the design of video decoders. This chapter details also the Dataflow Process Network MoC, adopted by the RVC standard, and its derived MoCs. The RVC-CAL language is presented later as the Domain Specific Language that translates an algorithm into a source code that respects the RVC MoCs. An RVC-CAL code needs, later, to be compiled to obtain a lower-level implementable representation in a software (C, Java) or hardware (VHDL, Verilog) code. Chapter 3 presents some existing tools for compiling RVC-CAL Dataflow programs: OpenDF and Orcc. This Chapter ends with the localization of the problematic of this thesis by showing the limitations of these tools in terms of hardware generation.

The first point to study is presented in chapter 4 and it concerns the validation of hardware implementations. We introduced a functional validation methodology that takes benefit of the software implementation generated with Orcc. Indeed, RVC-CAL is a target agnostic language and the software implementation associated

to an RVC-CAL design can be used to validate the correctness of the global algorithm. Moreover, the software debug speed in is much higher than the hardware one. This principle is applied on the high level RVC-CAL, then, after a manual transformation of the algorithm to low level, the software validation is used again because transforming high to low level involves the addition of algorithms, states and transitions that may generate new errors. Once the low level RVC-CAL is validated, the design is synthesizable using hardware compilers and, generally, the generated hardware implementation behaves correctly. This methodology showed very interesting results to accelerate the tough validation step. For the MPEG-4 Part 2 Simple Profile, we notice a gain in the conception time of more than 30%.

After the positive assessment of the functional verification, the last limitation of hardware generation from high level RVC-CAL is located in the transition from high to low level RVC-CAL that used to be manual before our work. The solution of improving the hardware compiler is quickly excluded since the source code of the compiler is very complicated and its intermediate representation is very low level and hard to be explored. Consequently, we considered Orcc compiler as a new and evolved compiler with a clear Intermediate Representation that allows easily adding automatic transformations. The objective was to use Orcc and generate an XLIM code directly synthesizable with OpenForge. After the elaboration of the XLIM back-end of Orcc by the Orcc team, we intervened by adding automatic transformation that detects the non-compliant features and transforms them while keeping the global behavior of the process. Later, this transformation was considerably optimized. This achievement has solved the main issue of the hardware generation flow from Dataflow programs. Indeed, the time we save when we develop applications using RVC-CAL can be lost during the process of manual modification from high to low level. The use of such automatic transformation keeps the important gain of 30% in the time to market. In addition, results show that the infrastructure generates an implementation with performances obviously lower than VHDL IPs but they remain close and acceptable.

We applied this automatic transformation on two different applicative contexts: the MPEG-4 Part 2 SP video decoder already present in the Video Tool Library of RVC and a still image codec called LAR that we developed during this thesis. This choice is considered because these two applications present available reference hardware implementations necessary for the assessment of our work. Chapter 6 presents an overview on the existing MPEG-4 decoders in the VTL of RVC. The RVC-CAL actors of the MPEG-4 part 2 SP decoder and MPEG-4 part 10 decoder are also illustrated. This chapter gives also a full comparative study between automatically generated, optimized and hardware reference decoders. The specific IDCT2D actor was tested separately to compare the transformation with an existing equivalent IP. The generated implementation of this actor is also compared with an automatic generation using an academic tool that transforms C to VHDL. The last Chapter 7, presents the LAR image coder principle and the design solutions we de-

veloped. We present also in this Chapter a comparative study between a VHDL IP and the automatically generated IP using our transformation.

8.2 Perspectives

The achievements of this thesis resulted in the resolution of the main limitations of the hardware generation flow using RVC standard. Currently, many other aspects of the developed automatic transformations are not yet tested.

▶ **Power consumption**
Till now, our validation stops at the correct simulation of of the implementation. It is necessary to study the impact of the transformations on the power consumption of the generated circuits and to achieve a comparison with reference IPs. Indeed, the way an algorithm is written may have an important impact on the power consumption. Such study may provide information on the optimal type or size of the created variables during the transformation step.

▶ **Toward more and more complex applications**
The VTL presents currently more complex designs such as the MPEG-4 AVC decoder which is much more complex than MPEG-4 part 2 Simple Profile decoder and it is possible that it reveals new cases of conflict or errors that have to be managed by the transformation. So it would be very important to test a hardware implementation of the MPEG-4 AVC FREXT profile as a more complex context.

With the experience we acquired through this thesis, it is possible to look for hardware implementation of the new emerging HEVC decoder. Currently, many parts of this decoder are in development with RVC specifications in IETR and EPFL. HEVC presents an important compression rate compared to MPEG-4 AVC while keeping approximatively the same complexity. This finding is very motivating since we already made a successful functional verification of the MPEG-4 FREXT profile. The objective would be to look for an efficient co-design implementation by finding a compromise between hardware and software available solutions.

▶ **From general to customized transformation**
Other improvements of the transformation can be added concerning the customization of the transformation process. For the moment, the transformation applies a general modification whatever the complexity of the actor. The use of internal buffers and indexes to store data is explained by the fact that many actions may

read from the same port. Every action has to read enough tokens and then check the guard condition. If this condition is wrong then the read data is lost. So the use of shared buffers resolves this problem. Consequently, the transformation can further be optimized by making more actor analysis to detect actors with simple FSM or actions reading exclusively from ports. These actors will be transformed into mono-token actions reading directly from the FIFOs. Moreover, the analysis can lead also to the integration of ping-pong algorithms for the SDF and CSDF actors that have the same number of tokens to read and to write. The ping-pong algorithm requires only to double the size of read/write internal buffers and a function that switches indexes from the first index to the half index and inversely.

► **Co-designing with Orcc**

In the field of ESLD, the solutions we present can open several perspectives for future works. Indeed, This transformation enables Orcc to generate mixed architectures. The hardware generation can be insured automatically using our transformation. For software generation, the Transport Triggered Architecture (TTA) processors are a potential solution. These processors have the particularity to apply computations as side effects of data transports which means that the data in the buses control the behavior of the processor. This property is totally compatible with the Dataflow Model of Computation we use in RVC. More interesting, a TTA based back-end [82] is in development by PhD student (IRISA/INSA of Rennes) and Orcc development team member Hervé Yviquel. The tests applied by Yviquel et.al on the MVG design of MPEG-4 part 2 SP decoder are successful and very promising.

The strength of this work lies in the combination of software validation and hardware implementation. The multiple Orcc backends open many perspectives of system enhancement. In addition to the TTA back-end, Orcc development team is improving the Promela back-end analyzing the schedulability, the LLVM back-end offering a software reconfiguration and the embedded C back-end targeting multi-core DSPs.

► **To finish**

More generally, we participated to resolve important limitations in the generation of hardware implementations from high level Dataflow programs. This work improved an existing compilation framework of RVC and can be used by any research group. All the approaches that we proposed are based on Opensource programs and softwares. Therefore researchers interested in this infrastructure can legally reuse the framework.

Appendix A

Frensh resume

Résumé en Français

► **Préface** Cette thèse entre dans le cadre d'une cotutelle entre l'Institut National des Sciences Appliquées de Rennes (INSA Rennes) et l'école Nationale d'Ingénieurs de Sfax (ENIS) de l'Université de Sfax. Le but des travaux de la thèse est de résoudre les problèmes existant de la génération hardware à partir des programmes flot de données et haut niveau.

► **Introduction et état de l'art** Les algorithmes de traitement de signal sont désormais de plus en plus complexes notamment dans le domaine du traitement de l'image et de la vidéo. L'évolution des algorithmes de compression ainsi que les architectures matérielles pouvant les supporter ont impliqué l'émergence de plusieurs standards de compression vidéo. Il faut noter que ces standards comprennent beaucoup de points communs mais les développeurs ne parviennent que difficilement à exploiter ces redondances de part la description monolithique des logiciels de référence associés à ces normes. Pour résoudre cette problématique, la norme RVC [55] a été introduite par la communauté MPEG pour répondre à ces limitations. Contrairement aux autres normes MPEG qui présentaient de la technologie de décodage (algorithmes, techniques de compressions), MPEG-RVC présente plutt une méthodologie de conception des systèmes de traitement du signal basée sur les modèles de calcul flot de données. Un design flot de données (voir Figure A.1) est un réseau de processus appelés acteur . Chaque acteur possède son propre état et il est complètement indépendant de l'état des autres acteurs du réseau. L'exécution d'un acteur est basée sur l'exécution des fonctions élémentaires appelées action .

Pour transformer le modèle présenté ci-dessus en un code fonctionnel, la norme MPEG-RVC a standardisé le langage CAL Actor Language (CAL) sous la norme MPEG-B [37]. Ce langage a été introduit lors du projet Ptolemy II à l'université de Berkeley. Ce langage est indépendant de la cible matérielle (hardware ou soft-

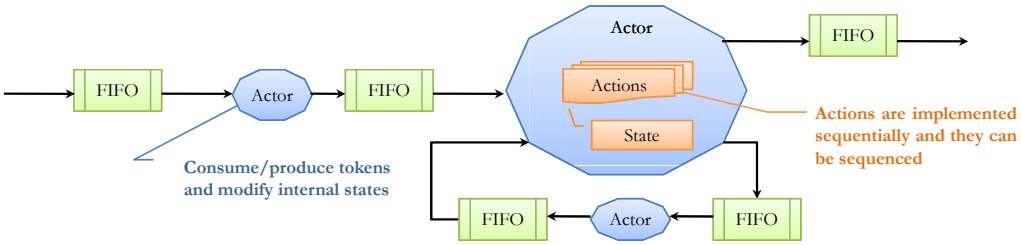

Figure A.1: Le modèle RVC

ware) et répond par conséquent à la méthodologie de conception ESLD (Electronic System Level Design) [58]. Cette méthodologie consiste à monter en niveau d'abstraction lors du développement d'une norme en utilisant des langages de domaine spécifique qui seront par la suite synthétisés automatiquement dans différents langages cibles (hardware ou software) puis compilés pour le compilateur de la cible visée. Pour le CAL, la norme RVC propose tout un framework pour la conception, la validation et la compilation des programmes. Nous étudions plus particulièrement dans ce travail l'outil de génération hardware appelé OpenForge [34]. Ce dernier prend en entrée (front-end) des acteurs CAL et un réseau, puis génère dans une première étape une description sous la forme Single Static Assignment (SSA) [19] proche des descriptions hardwares sous forme de fichier représenté sous une forme XML appelée Xlim [4]. Dans une deuxième étape, il transforme le code Xlim généré en Verilog synthétisable (voir Figure A.2).

La limitation d'OpenForge vient du fait qu'il n'est pas capable de synthétiser certaines structures de haut niveau du langage CAL. Le but de cette thèse est donc de trouver des solutions pour résoudre les problèmes de génération hardware des programmes flot de données haut niveau et aussi de trouver des méthodes rapides pour la validation des designs générés. Dans la suite de ce document, nous reviendrons sur l'explication de l'essentiel du langage CAL, nous préciserons les structures non compilables avec OpenForge. Dans la partie des réalisations, nous expliquerons une méthodologie de conception et de validation rapide des programmes CAL et nous détaillerons le processus de transformation des structures non compilables. Les réalisations étant testées sur des exemples d'applications de traitement vidéo et image, nous présenterons l'essentiel des résultats d'implémentation du décodeur MPEG4 Simple Profile Part 2 et du codec d'images fixes LAR.

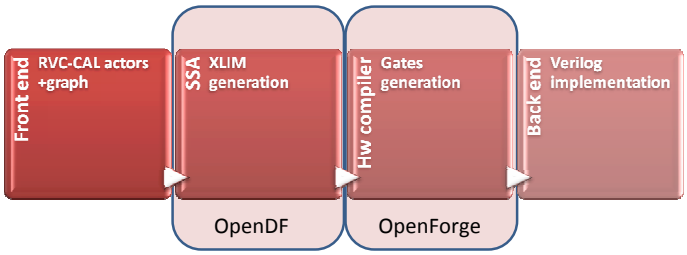

Figure A.2: Etapes de compilation de l'outil OpenForge

A.1 Le modèle de calcul flot de données

La norme RVC est basée sur le modèle de calcul DPN (Dataflow Process Network) [53]. Ce modèle fixe le comportement des acteurs de faon à ce que chaque exécution est relative à l'exécution d'une action. Toute action est déclenchée suite à la validation de sa règle de tir ou firing rule . La règle de tir précise le nombre de jetons que l'action demande pour s'exécuter ce qui en résulte que le fonctionnement de l'acteur globalement est dépendant seulement de la présence des jetons suffisants pour exécuter ses actions. Un acteur est défini de la syntaxe suivante où on définit son tag, ses entrées/sorties et leur dynamique :

```
1 actor example (int m)
2 int (size=8) IN1, uint (size=12) IN2 ==> int(size=13) OUT :
3
4   // algorithm
5
6 end
```

Une action est définie comme suit :

La précision des ports utilisés par l'action et le nombre de jetons à consommer sont définis dans la syntaxe (ligne 4). Si le nombre de jetons consommés est inférieur ou égal à 1 alors on appelle l'action action mono-token sinon on l'appelle action multi-token . Un acteur peut éventuellement contenir une machine d'état finie (FSM) pour gérer certains cas ne pouvant pas être décrits avec seulement des ap-

```
actor sum ()
(int size=8) INPUT1, (int size=8) INPUT2 ==> int(size=8) OUTPUT:

 add: action INPUT1:[ i1 ], INPUT2:[i2] ==> OUTPUT:[s]
 var
   int s
 do
   s:= i1 + i2 ;
 end
end
```

pels d'actions. La machine d'état de la Figure A.3 est présentée par la syntaxe CAL suivante :

```
schedule fsm init_state:
  init_state      ( action1 ) --> state10;
  state10         ( action2 ) --> state10;
  state10         ( action3 ) --> state11;
  state11         ( action4 ) --> init_state;
end
```

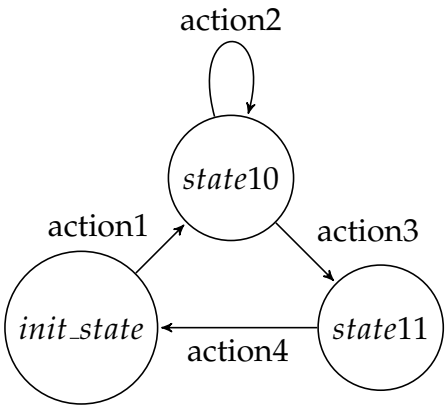

Figure A.3: Exemple d'un graphe d'une FSM

Si deux actions ont leurs règles de tirs validées en même temps, le langage CAL n'en sélectionnera qu'une. Il propose des priorités pour fixer l'action à choisir en cas de conflit et éviter par la suite un choix aléatoire qui impliquerait un comportement non déterministe. Il existe aussi la notion des actions non tagguées. Ces actions n'ont pas de tag et ne Figurent pas dans la machine d'états. Leur propriété la plus importante est qu'elles sont prioritaires en exécution par rapport à toutes les autres actions. Par conséquent, si l'acteur est dans un état courant et que l'action non tagguée et validée, cette action sera exécutée ensuite l'acteur revient à sont état et continue les transitions de la FSM. La problématique de génération de code hardware est essentiellement liée à la forme multi-token des actions. En effet, les multi-read ne peuvent être instanciés en hardware à l'image de ce qui est possible de réaliser simplement en software. Par conséquent il est nécessaire de transformer du code multi-token aboutissant ainsi à du CAL bas niveau. Notre objectif est donc

de définir le mécanisme de transformation automatique. L'exemple suivant montre dans la partie (a) un code multi-token et son équivalent en mono-token dans la partir (b). L'idée de la transformation de code est de créer une action mono-token pour la lecture et la faire boucler dans un état read le nombre de read nécessaire de fois pour arriver au nombre souhaité de jetons. Ce ci permet ensuite de passer à l'exécution du code du corps de l'action lorsqu'une action possède une règle de tir valide.

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 add: action IN:[ i ] repeat 5
 ==> OUT:[ s ]
 var
   int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + i[k] ;
   end
 end
end
```

(a)

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 List (type: int (size=8), size = 5) data;
 int counter :=0 ;

 read: action IN:[ i ] ==>
 do
   data[counter] := i ;
   counter := counter + 1 ;
 end

 read_done: action ==>
 guard
   counter = 5
 do
   counter := 0 ;
 end

 process: action ==> OUT:[ s ]
 var
  int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + data[k] ;
   end
 end

 schedule fsm state0:
   state0 (read)      --> state0;
   state0 (read_done) --> state1;
   state1 (process)   --> state0;
 end

 priority
   read_done > read;
 end

end
```

(b)

A.2 La méthodologie de validation rapide

Pour transformer le code CAL du haut vers le bas niveau, nous avons proposé une transformation automatique dans le cur d'un compilateur des programmes CAL.

Nous avons choisi un compilateur en cours de développement au sein de l'équipe IETR à l'INSA appelé Orcc [41]. Cet outil est capable de générer plusieurs langages à l'aide de back-ends (un par langage) pour un même code CAL initial comme présenté dans la Figure A.4. Orcc parse le CAL et crée une représentation intermédiaire (IR) qui dérive d'un arbre abstrait de type AST (Abstract Syntax Tree).

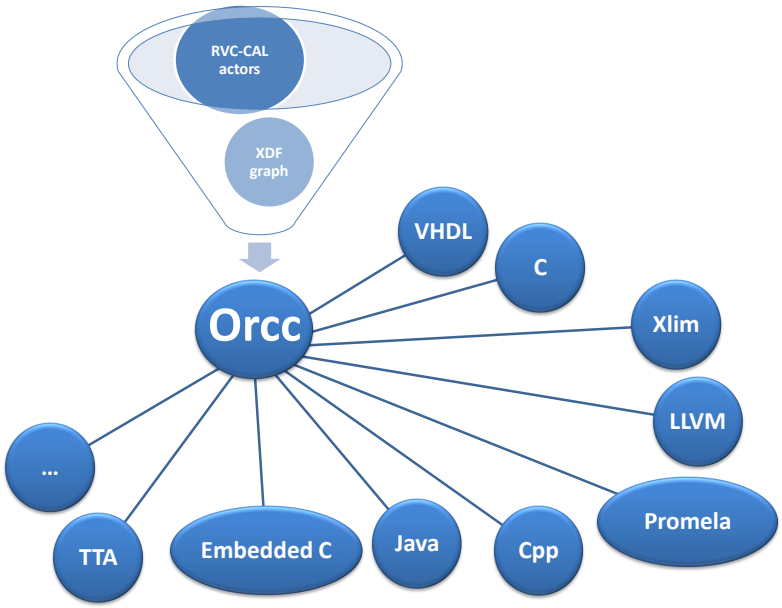

Figure A.4: Les back-ends de Orcc

Pour arriver à valider rapidement un design durant la conception de l'implémentation hardware, nous avons proposé une méthodologie fonctionnelle de validation [45]. Le but est de valider à chaque étape du flot de conception. Le design commence par une description haut niveau permettant l'obtention d'une solution le plus rapidement possible. Ce design sera validé sur une plateforme software en générant un code C à partir du CAL via l'outil Orcc. Une fois le design haut niveau validé, une transformation du code en mono-token est réalisée. Le design obtenu est de même compilé en C et validé par la suite avec un debugger C. Finalement le design mono-token est utilisé avec OpenForge pour obtenir une implémentation hardware. Voir résumé de la méthodologie dans la Figure A.5.

L'avantage de cette démarche est qu'elle nous laisse un maximum de temps de conception dans un contexte fonctionnel software. Le débogage dans une plateforme software est nettement plus rapide que celui dans un synthétiseur hardware. En outre, cette méthodologie permet de limiter beaucoup d'erreurs qui mènent à l'échec de la génération hardware avec OpenForge. Nous avons démontré avec des tests sur des design CAL du LAR que cette méthodologie nous assure un gain de 40

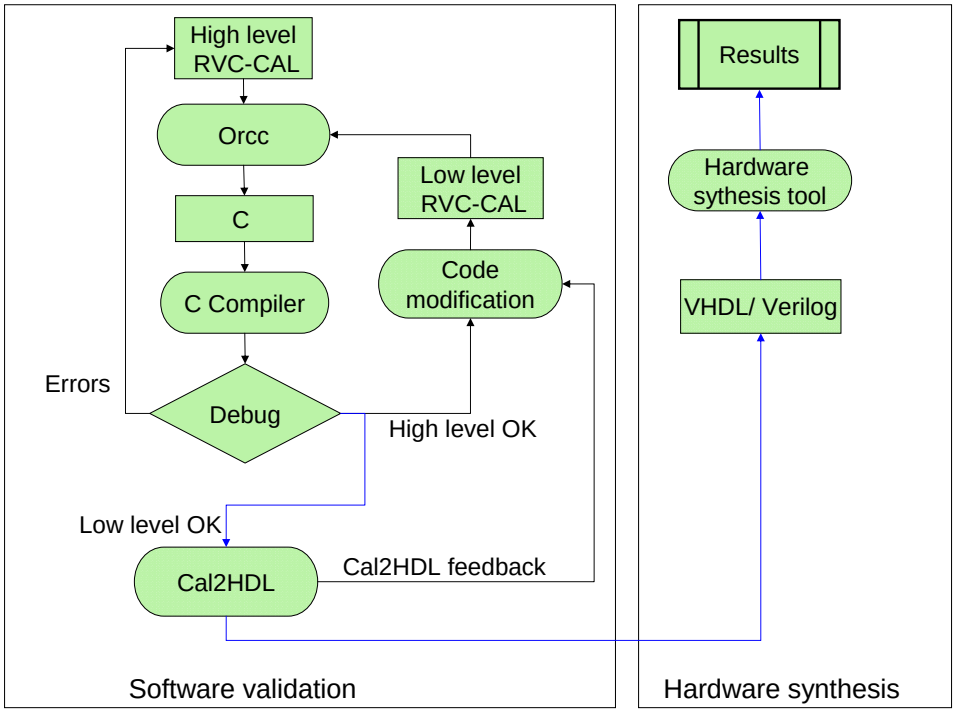

Figure A.5: Principe de la méthodologie

A.3 La transformation automatique du code

La phase de transformation du code du haut vers le bas niveau reste le goulot d'étranglement majeur de la synthèse haut niveau des programmes flot de données. Nous avons donc pensé à automatiser cette transformation. Dans les outils existants, Orcc est un compilateur capable de générer du Xlim. Ce dernier représente le front-end d'OpenForge. L'idée de notre contribution est de faire une transformation dans le cur de Orcc pour générer un code Xlim compatible et synthétisable avec OpenForge comme présenté dans la Figure A.6.

Le processus de transformation [43] [47] commence par la détection des lectures multi-token. Une fois trouvées, on supprime l'input pattern de l'action pour qu'il ne comporte plus de lecture. Une action non tagguée est par la suite créée pour faire des lectures mono-token comme présenté dans le code suivant où on remarque que l'action non tagguée est en train de lire un seul jeton pour le mettre dans un buffer (Figure A.7).

Le buffer en question est lui aussi créé durant la phase de transformation en lui associant les deux indexes IdxW et IdxR qui s'incrémentent respectivement lors d'une lecture d'un jeton et lors de la fin des lectures des jetons (voir Figure A.8).

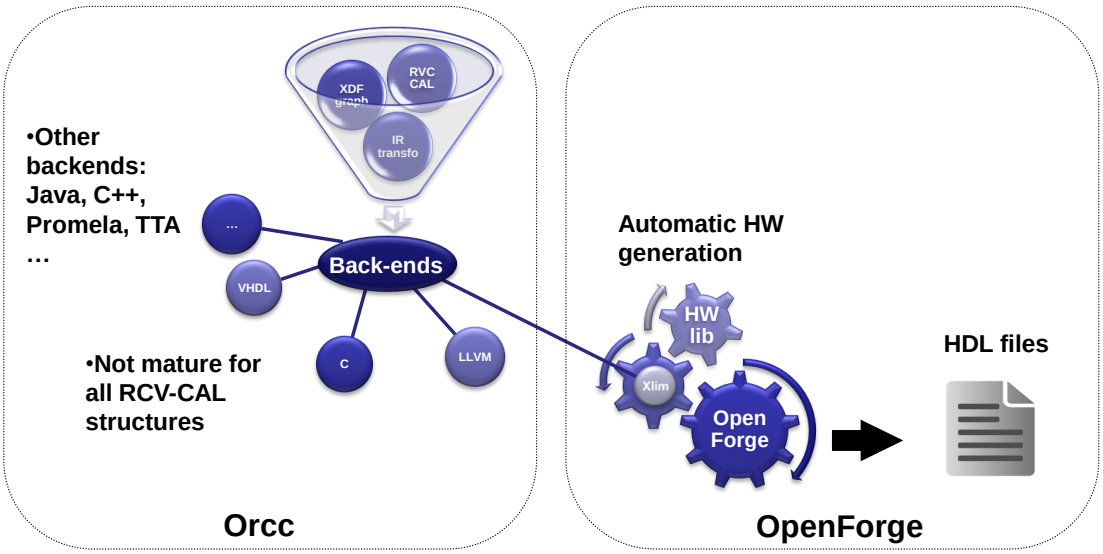

Figure A.6: Nouveau flot de conception avec Orcc

```
1  action fifo1:[t] ==>
2  guard
3          idxW - idxR < size
4  do
5          buffer[idxW] := t;
6          idxW := idxW + 1;
7  end
```

Figure A.7: Exemple d'une action non-taggée

Le buffer est aussi conu d'une faon circulaire ce qui veut dire que lorsque la dernière case est occupée il continue à enregistrer dans la première case. Pour cela nous avons ajouté des modulos pour gérer les indexes de lecture et d'écriture. Bien évidemment, le buffer ne doit pas enregistrer une nouvelle valeur dans une case comportant une donnée qui n'est pas encore utilisée sinon elle sera à jamais écrasée et perdue. C'est ce qui explique la condition guard (IdW-IdxR ¡ size) dans le code de l'action non tagguée. Le code de l'action sum4 de la Figure A.9.a est remplacé donc par le code la Figure A.9.b :

Pour les écritures multi-tokens, la solution retenue est de créer un état dans la machine d'états dans lequel l'acteur bloque pendant l'exécution d'action d'écriture mono-token. Une autre action détecte que le nombre de jetons écrits est atteint et change l'état de l'acteur pour exécuter d'autres actions. Dans la représentation intermédiaire de Orcc nous créons les actions les actions mono-token d'écriture, les tableaux et indexes nécessaires ainsi que les transitions dans la FSM. L'action de la Figure A.10.a est replacée par le code de la Figure A.10.b et la machine d'état est présentée dans la Figure A.11.

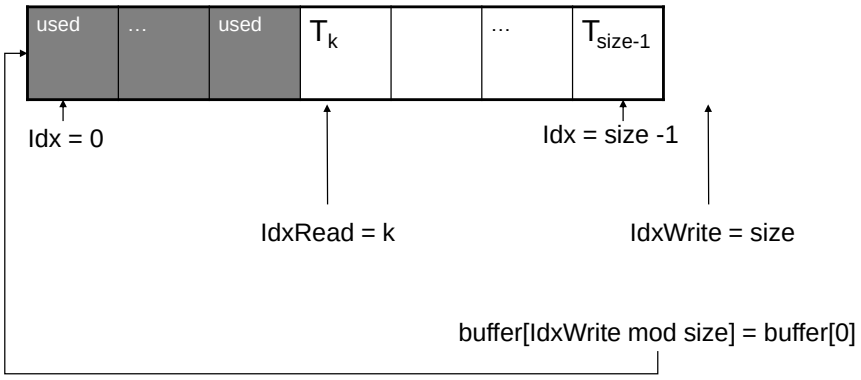

Figure A.8: structure du buffer créé avec ses indexes d'écriture et de lecture

```
actor sum4 () int (size=8) IN
==> int(size=10) OUT:

 add: action IN:[ i ] repeat 4
 ==> OUT:[ s ]
 var
   int s := 0
 do
   foreach int k in 0 .. 3 do
     s := s + i[k] ;
   end
 end
end
```

(a) Action multi-token à transformer

```
actor sum4 () int (size=8) IN
==> int(size=10) OUT:

 List (type: int (size=8), size = 5) data;
 int index := 0;

 action IN:[ i ] ==>
 guard
        index < 4
 do
   data[index] := i ;
   index := index + 1 ;
 end

 add: action ==> OUT:[ s ]
 var
  int s := 0
 do
   foreach int k in 0 .. 3 do
     s := s + data[k] ;
   end
 end

end
```

(b) action mono-token après transformation

Figure A.9

Si l'acteur ne présente pas une FSM, on crée un état init dans le quel on boucle toutes les actions. Ensuite on ajoute le macro bloc d'écriture de la manière suivante (Figure A.12):

Si une FSM existe déjà (Figure A.13) alors on insère le macro bloc comme indiqué dans la Figure A.14.

Après avoir terminé la transformation, il était nécessaire de réaliser d'autres transformations qui remplacent les tableaux locaux dans les actions par des tableaux

```
1
2  actor duplic4 () int (size=8) IN
3  ==> int(size=8) OUT:
4
5   duplicata: action IN:[ i ]
6   ==> OUT:[ tab ] repeat 4
7   var
8     List (type: int, size = 4) tab;
9   do
10    foreach int k in 0 .. 3 do
11      tab[k] := i;
12    end
13  end
14 end
```

```
1  actor sum4 () int (size=8) IN
2  ==> int(size=8) OUT:
3
4   List (type: int (size=8), size = 5) tab;
5   int counter := 0;
6
7   duplicata: action IN:[ i ] ==>
8   do
9     foreach int k in 0 .. 3 do
10      tab[k] := i;
11    end
12  end
13
14  write: action ==> OUT:[ s ]
15  var
16   int s := 0
17  do
18      s := tab[counter];
19      counter := counter + 1;
20  end
21
22  write_done: action ==>
23  guard
24    counter = 4
25  do
26    counter := 0;
27  end
28 end
```

Figure A.10:
(a) Action d'écriture multi-token à transformer
(b) Action d'écriture multi-token transformée

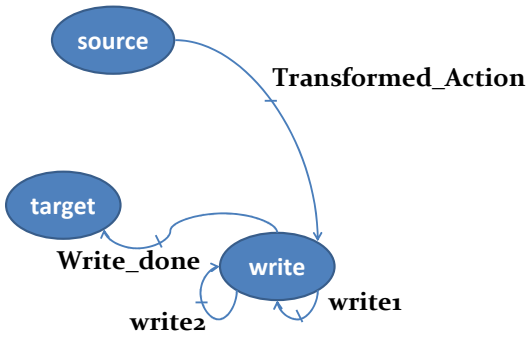

Figure A.11: Macro bloc FSM créé pour les écritures multi-token

en tant que variables d'état. Ceci est d au fait que les générateurs hardware ne supportent pas des listes locales. Nous avons aussi réalisé une nouvelle transformation qui transforme une division entière en une division euclidienne compatible avec les opérations réalisables en hardware. Nous avons aussi utilisé la représentation intermédiaire de Orcc pour créer des testbenchs automatiques en utilisant des méthodes de pretty printing tel que le générateur de code Xtend ou string template . Ce test bench pointe directement sur des vecteurs de test créés lors des simulations software.

A.4 Application sur le décodeur vidéo MPEG 4 SP Part 2

Pour tester notre méthodologie nous avons choisi comme premier contexte applicatif le décodeur MPEG4 Simple Profile Part 2 [39]. Le choix est expliqué par le fait que ce décodeur est parmi les architectures les plus stables dans la librairie RVC. De plus nous avons des designs de référence notamment une IP VHDL et une description bas niveau orientée hardware et développée par Xilinx. Le design du décodeur est présenté dans la Figure A.15. Il est composé de trois étages de décodage pour chaque composante luminance et chrominances pour la texture et la compensation du mouvement.

Après avoir utilisé notre méthodologie, nous avons obtenu les résultats des

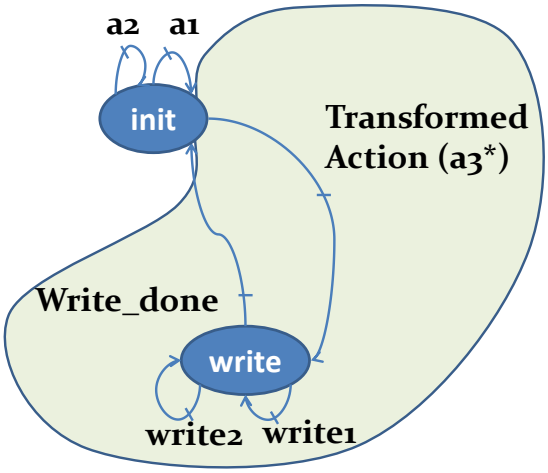

Figure A.12: Bloc FSM ajouté à l'état init

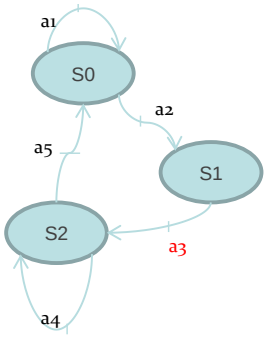

Figure A.13: FSM initiale avec action a3 multi-tokens

tableaux suivants sur une FPGA virtex 4 xc4vlx160-12ff1148:

	implémentation
Slice Flip Flops	13,575/135,168 (10%)
Occupied Slices	18,178/67,584 (26%)
4 input LUTs	34,333/135,168 (25%)
FIFO16/BRAM16s	14/288 (4%)
Bonded IOBs	107/768 (13%)

Nous remarquons certes une différence de 75Pour se comparer aussi par rapport aux outils existants, nous avons choisi l'outil académique GAUT [18] [17] qui est un générateur automatique de code hardware à partir du C et qui est très performant sur les architectures de traitement massif des données. GAUT étant incapable de transformer un design au niveau système, nous avons choisi de comparer seulement

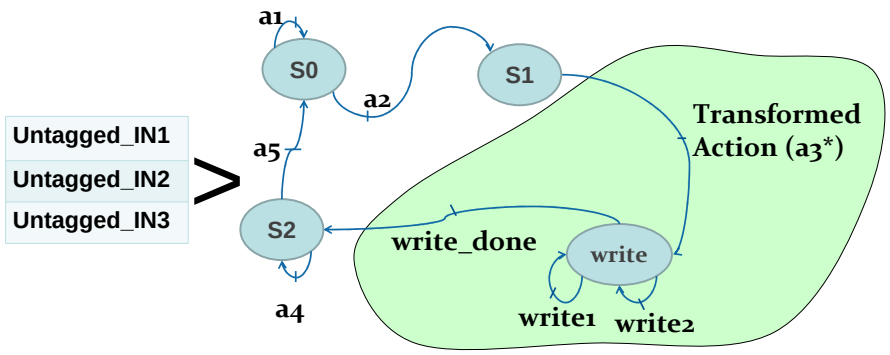

Figure A.14: FSM résultante après l'ajout du macrobloc

	performances
Maximum frequency (MHz)	26.67
Latency (μs)	306,4
Cadency (MHz)	2.33
Processing time (ms/image)	11,01
Throughput frequency (MHz)	2,2
Global image processing (FPS)	90,82

La comparaison avec l'IP VHDL et le design Xilinx est présentée dans le tableau suivant:

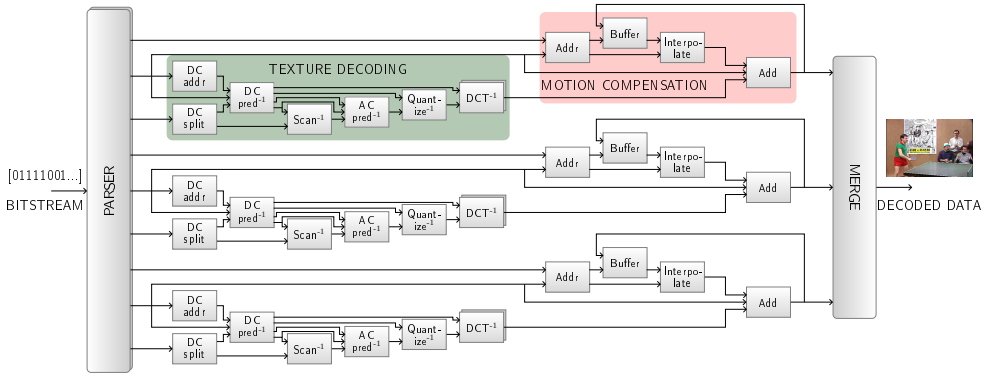

Figure A.15: Architecture du design du décodeur MPEG 4 SP Part 2

	Design automatique	VHDL IP	CAL
Slice	18,178 (26%)	4637 (7%)	3872 (6%)
LUT	34,33 (25%)	7923 (6%)	7720 (6%)

les performances sur l'IDCT2D dans le tableau suivant :

	Design GAUT	Design transformé
Slice Flip Flops	2,080/135,168 (2%)	1,988/135,168 (2%)
Occupied Slices	2,477/67,584 (3%)	2,353/67,584 (3%)
4 input LUTs	4,243/135,168 (3%)	4,458/135,168 (3%)
Bonded IOBs	627/768 (81%)	49/768 (6%)

Nous remarquons que les résultats obtenus en terme de surface sont meilleures que ceux générés par GAUT sur tous les critères.

A.5 Application sur le codec d'images fixes LAR

Un autre contexte applicatif choisi pour expérimenter les travaux de cette thèse qui est le codec d'images fixes LAR Locally Adaptive Resolution [23]. Ce codec se compose de deux parties essentielles : le codage/décodage spatial et le codage/décodage de texture voir Figure A.16.

L'originalité du LAR est qu'il présente un codage adaptatif selon l'activité sur l'image. Donc au lieu de décomposer l'image sur un ensemble de blocs uniformes comme JPEG par exemple, le LAR va décomposer l'image en appliquant un gradient morphologique qui détecte l'activité sur l'image et considère des blocs de grande taille quand l'activité est faible et des blocs de petite taille quand l'activité est importante (voir Figure A.17). Ce ci est appliqué utilisant un processus de décomposition de l'image appelé Quadtree .

Nous avons développé en CAL une baseline du LAR composée d'une partie de codage spatial appelé aussi FLAT LAR (Figure A.18) et une partie de codage et décodage de texture basée sur la transformée Hadamard (Figure A.19).

Le LAR est synthétisé en hardware en utilisant la méthodologie de transformation automatique du code et nous présentons les résultats de comparaison entre le design transformé manuelle et celui transformé automatiquement.

Les résultats montrent une grande similitude en consommation de surface. La différence en termes de performances temporelles est tout à fait logique vu la manière optimisée avec la quelle le design manuel a été réalisé. Cependant on note que cette méthodologie est trois fois plus rapide en termes de temps de conception.

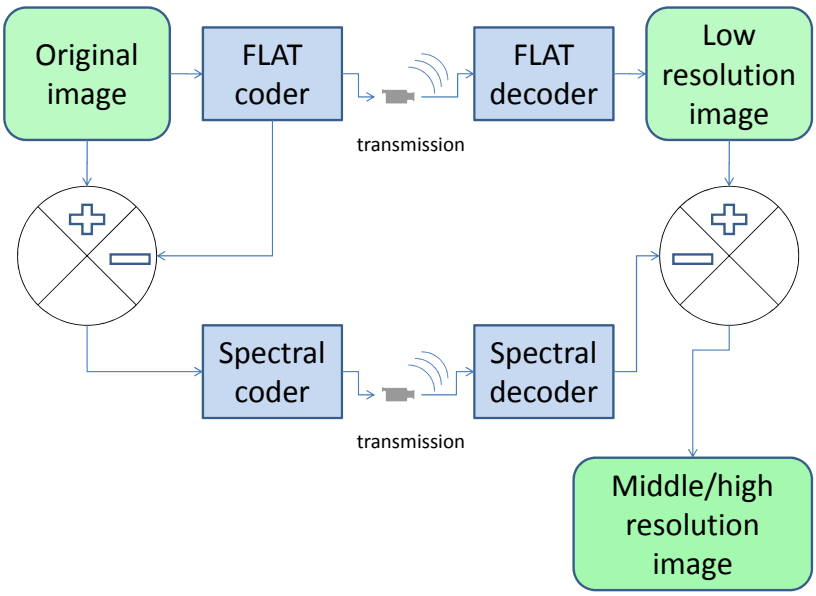

Figure A.16: Schémas de codage et décodage du LAR

Transformation	Automatique	Manuelle
Slice Flip Flops	20,452/135,168 (15%)	12,157/135,168 (8%)
Occupied Slices	47,576/67,584 (70%)	43,602/67,584 (67%)
4 input LUTs	59,868/135,168 (44%)	53,417/135,168 (39%)
Bonded IOBs	41/768 (5%)	41/768 (5%)

Transformation	Automatique	Manuelle
Development time	30%	100%
Maximum frequency (MHz)	61,43	85,27
Latency (ms)	0,42	0,12
Throughput frequency (MHz)	3,5	5,6
Processing time (ms/image)	35	19
Global image processing (FPS)	34	53

A.6 Conclusion et perspectives

Cette thèse a pour objectif d'automatiser la génération des architectures hardware à partir des programmes flot de données. Le but était de garder un niveau d'abstraction assez élevé pour rester dans le niveau système et satisfaire par conséquent le contexte ESLD. Nous avons présenté les modèles flot de données ainsi que le framework RVC. Par la suite nous avons localisé le goulot d'étranglement

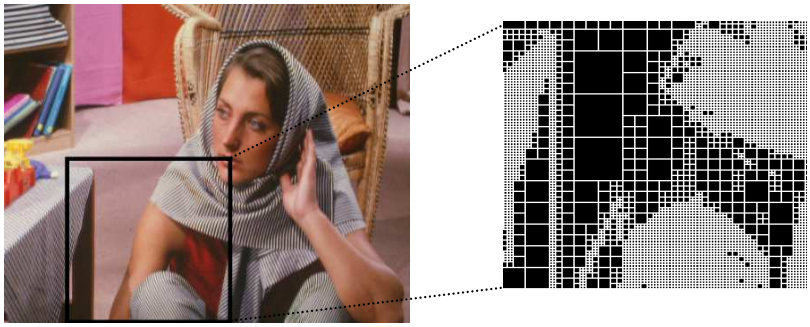

Figure A.17: Image des tailles résultante du gradient morphologique

dans le flot de génération hardware et nous avons développé une transformation automatique du code dans le cur du compilateur Orcc pour détecter les structures non compilables et les transformer en leur équivalent de structures compilables tout en gardant le même comportement global de l'application. Les travaux comportent aussi une méthodologie de conception rapide des systèmes flot de données avec une approche de validation fonctionnelle qui valide une grande partie de la conception sur une plateforme logicielle rapide et plus facile à déboguer. Les méthodologies ont été finalement appliquées sur des applications de traitement video et image et des études comparatives ont été réalisées pour montrer l'intérêt de cette recherche. Dans nos futurs travaux nous comptons appliquer la transformation du code sur des applications plus complexes de traitement vidéo tel que MPEG AVC et HEVC. Il est aussi possible de créer une transformation personnalisée puisque l'actuelle transformation est générale sur tous les acteurs et dans des cas simples d'acteur statiques par exemple il est préférable d'optimiser le circuit avec une transformation moins compliquée mais dédiée. La possibilité de faire du Co-design est une autre perspective à nos travaux surtout avec l'apparition d'un nouveau back-end de Orcc qui génère une implémentation vers des processeurs TTA (softcores embarqués dans un FPGA) qui ont une architecture qui répond parfaitement au modèle de calcul flot de données. Nous notons que tous les codes de notre recherche sont libres de droit comme tous les outils utilisés lors de cette thèse ce qui offre la possibilité à tous les chercheurs de profiter de notre étude.

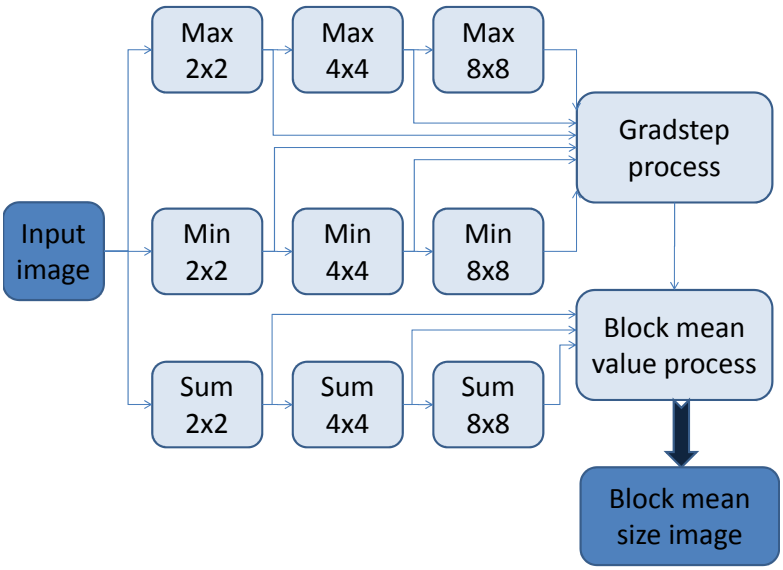

Figure A.18: Architecture développée pour le codage spatial

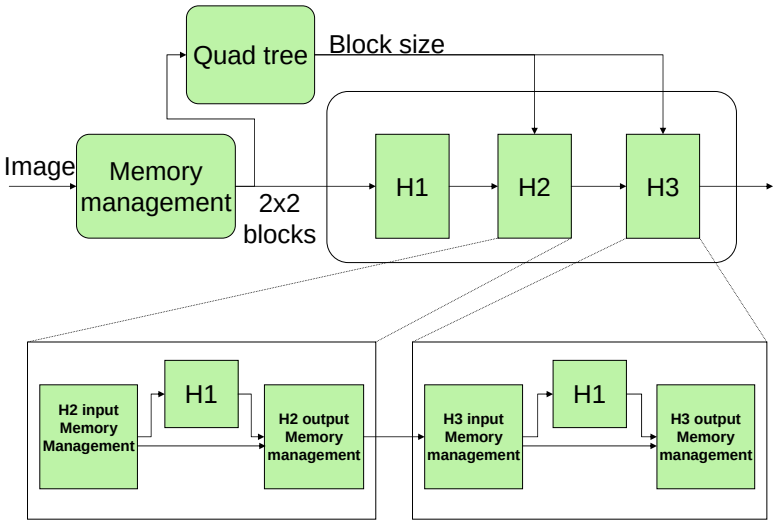

Figure A.19: Architecture développée pour le codage de fréquence

List of Figures

List of Tables

Personal publications

- Khaled Jerbi, Mickaël Raulet, Olivier Déforges, and Mohamed Abid. "Design of an Embedded Low Complexity Image Coder using CAL language". *DASIP 2009 proceedings*, September 2009.
- Khaled Jerbi, Matthieu Wipliez, Mickaël Raulet, Marie Babel, Olivier Déforges, and Mohamed Abid. "Fast hardware implementation of an Hadamard transform using RVC-CAL dataflow programming". In *proceedings of IEEE EMC*, 9 2010.
- Khaled Jerbi, Matthieu Wipliez, Mickaël Raulet, Marie Babel, Olivier Déforges, and Mohamed Abid. "Automatic method for efficient hardware implementation from RVC-CAL dataflow: A LAR coder baseline case study". *Journal Of Convergence*, 1(1):8592, 12 2010.
- Khaled Jerbi, Mickaël Raulet, Olivier Déforges, and Mohamed Abid. "Automatic generation of synthesizable hardware implementation from high level RVC-CAL design". In *ICASSP 2012: Proceedings of the 37th International Conference on Acoustics Speech and Signal Processing*, pages 15971600, 2012.
- Khaled Jerbi, Mickaël Raulet, Olivier Déforges, and Mohamed Abid. "Automatic generation of optimized and synthesizable hardware implementation from high level dataflow programs". *VLSI Design*, 2012:14, 2012. 10.1155/2012/298396.
- Khaled Jerbi, Tarek Ouni, and Mohamed Abid. "A dataflow description of acc-jpeg coder". In *the proceedings of 2012 International Conference on Signal Processing and Multimedia Application (SIGMAP 2012)*.

References

[1] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transfom. *IEEE Transactions on Computers*, 23:90–93, 1974.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: principles, techniques, and tools. *Reading, MA,*, 1986.

[3] B. Bailey, G.E. Martin, and A. Piziali. *ESL design and verification: a prescription for electronic system-level methodology*. The Morgan Kaufmann series in systems on silicon. Morgan Kaufmann, 2007.

[4] E. Bezati, H. Yviquel, M. Raulet, and M. Mattavelli. A unified hardware/-software co-synthesis solution for signal processing systems. In *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pages 1 –6, nov. 2011.

[5] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for dsp systems. In *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, RSP '00, pages 84–, Washington, DC, USA, 2000. IEEE Computer Society.

[6] Bishnupriya Bhattacharya, Shuvra S. Bhattacharyya, and Senior Member. Parameterized Dataflow Modeling for DSP Systems. *IEEE Transactions on Signal Processing*, 49:2408–2421, 2001.

[7] S Bhattacharyya, G Brebner, J Eker, J Janneck, M Mattavelli, C von Platen, and M Raulet. OpenDF - A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems. First Swedish Workshop on Multi-Core Computing, MCC , Ronneby, Sweden, November 27-28, 2008, 2008.

[8] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397 –408, feb 1996.

[9] B Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11:14–24, August 1986.

[10] J. Boutellier, C. Lucarz, S. Lafond, V.M. Gomez, and M. Mattavelli. Quasi-static scheduling of CAL actor networks for reconfigurable video coding. *Journal of Signal Processing Systems*, pages 1–12, 2008.

[11] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng (eds.). PtolemyII - heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemyII). Technical Memorandum UCB/ERL M04/27, University of California, Berkeley, CA USA 94720, July 2004.

[12] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation*, 4:155–182, 1994.

[13] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 1:429–432, 1993.

[14] Chang-Hsuan Chang, Ming-Hung Chang, and Wei Hwang. A flexible two-layer external memory management for h.264/avc decoder. In *SOC Conference, 2007 IEEE International*, pages 219 –222, sept. 2007.

[15] C. Click. Global code motion/global value numbering. *ACM SIGPLAN Notices*, 30(6):246–257, 1995.

[16] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe. Quasi-static scheduling of independent tasks for reactive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1492–1514, 2005.

[17] P. Coussy, D.D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *Design Test of Computers, IEEE*, 26(4):8 –17, july-aug. 2009.

[18] Philippe Coussy, C. Chavet, Pierre Bomel, D. Heller, E. Senn, and E. Martin. GAUT: A High-Level Synthesis Tool for DSP applications. In Philippe Coussy & Adam Morawiec, editor, *High-Level Synthesis: From Algorithm to Digital Circuits*, pages 147–170. Springer, June 2008.

[19] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):490, 1991.

[20] D. Dearman, A. Cox, and M. Fisher. Adding control-flow to a visual data-flow representation. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 297 – 306, may 2005.

[21] Olivier Déforges and Marie Babel. Lar method: from algorithm to synthesis for an embedded low complexity image coder. *IEEE 3rd International Design and Test Workshop*, 2008.

[22] Olivier Déforges and Marie Babel. LAR method: from algorithm to synthesis for an embedded low complexity image coder. *IEEE 3rd International Design and Test Workshop, IDT'08*, 2008.

[23] Olivier Déforges, Marie Babel, Laurent Bédat, and Joseph Ronsin. Color LAR Codec: A Color Image Representation and Compression Scheme Based on Local Resolution Adjustment and Self-Extracting Region Representation. *IEEE Trans. Circuits Syst. Video Techn.*, 17(8):974–987, 2007.

[24] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*.

[25] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

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

[27] Johan Eker and Jörn W. Janneck. An introduction to the Caltrop actor language, September 2001.

[28] R. Ernst, J. Henkel, and T. Benner. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Des. Test*, 10:64–75, October 1993.

[29] ETH Zrich. *Moses project: `http://www.tik.ee.ethz.ch/~moses/`*.

[30] S. W. Golomb. Run length codings. *IEEE Transactions on Information Theory*, pages 12(7): 399–401, 1966.

[31] J. Gorin, M. Wipliez, J. Piat, F. Preteux, and M. Raulet. An LLVM-based decoder for MPEG reconfigurable video coding. In *Signal Processing Systems (SIPS), 2010 IEEE Workshop on*, pages 81 –86, oct. 2010.

[32] J. Gorin, M. Wipliez, F. Preteux, and M. Raulet. LLVM-based and scalable MPEG-RVC decoder. *Journal of Real-Time Image Processing*, pages 1–12, 2010. 10.1007/s11554-010-0169-2.

[33] Steven Greenbaum and Stanley Jefferson. A compiler for hp vee. *Hewlett Packard Journal*, 49:98–99, 1998.

[34] Ruirui Gu, Jörn W. Janneck, Shuvra S. Bhattacharyya, Mickaël Raulet, Matthieu Wipliez, and William Plishker. Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis. *Circuits and Systems for Video Technology, IEEE Transactions on*, 19(11):1646–1657, 11 2009.

[35] IEEE Std 1076-1993. *IEEE Std 1076 - IEEE Standard VHDL Language Reference Manual*, 1993.

[36] IEEE Std 1364-2001. *IEEE Std 1364-2001 - IEEE Standard verilog Language Reference Manual*, 2001.

[37] ISO/IEC FDIS 23001-4: 2009. Information Technology - MPEG systems technologies - Part 4: Codec Configuration Representation, 2009.

[38] ISO/IEC FDIS 23002-4: 2009. Information Technology - MPEG video technologies - Part 4: Video tool library, 2009.

[39] J. Janneck. Notes on an actor language, 7th Ptolemy Miniconference. Technical report, University of California at Berkeley, February 2007.

[40] Jörn Janneck, Ian Miller, David Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. Synthesizing hardware from dataflow programs. *Journal of Signal Processing Systems*, pages 1–9, 2009. 10.1007/s11265-009-0397-5.

[41] Jörn W. Janneck, Marco Mattavelli, Mickaël Raulet, and Matthieu Wipliez. Reconfigurable video coding a stream programming approach to the specification of new video coding standards. In *MMSys '10: Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 223–234, New York, NY, USA, 2010. ACM.

[42] Khaled Jerbi, Tarek Ouni, and Mohamed Abid. A dataflow description of acc-jpeg coder. 2012. Accepted for the proceedings of 2012 International Conference on Signal Processing and Multimedia Application (SIGMAP 2012).

[43] Khaled Jerbi, Mickaël Raulet, Olivier Déforges, and Mohamed Abid. Automatic generation of synthesizable hardware implementation from high level RVC-CAL design. In *ICASSP'12: Proceedings of the 37th International Conference on Acoustics Speech and Signal Processing*, pages 1597–1600, 2012.

[44] Khaled Jerbi, Mickaël Raulet, Olivier Déforges, and Mohamed Abid. Design of an Embedded Low Complexity Image Coder using CAL language. *DASIP 2009 proceedings*, September 2009.

[45] Khaled Jerbi, Matthieu Wipliez, Mickaël Raulet, Marie Babel, Olivier Déforges, and Mohamed Abid. Automatic method for efficient hardware implementation from RVC-CAL dataflow: A lar coder baseline case study. *Journal Of Convergence*, 1(1):85–92, 12 2010.

[46] Khaled Jerbi, Matthieu Wipliez, Mickaël Raulet, Marie Babel, Olivier Déforges, and Mohamed Abid. Fast hardware implementation of an hadamard transform using RVC-CAL dataflow programming. *In proceedings of IEEE EMC*, 9 2010.

[47] Khaled Jerbi, Matthieu Wipliez, Mickaël Raulet, Marie Babel, Olivier Déforges, and Mohamed Abid. Automatic generation of optimized and synthesizable hardware implementation from high level dataflow programs. *VLSI Design*, 2012:14, 2012. 10.1155/2012/298396.

[48] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[49] The SISAL language web site. http://www.physics.nmt.edu/raymond/software/sisal/sisal.html.

[50] D. Lau, O. Pritchard, and P. Molson. Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 45 –56, april 2006.

[51] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.

[52] Edward A. Lee. A denotational semantics for dataflow with firing, January 1997.

[53] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[54] M.A. Losada and K.T. Mullen. The spatial tuning of chromatic mechanisms identified by simultaneous masking. *Vision Research*, 34(3):331 – 341, 1994.

[55] Marco Mattavelli, Jörn W. Janneck, and Mickaël Raulet. MPEG Reconfigurable Video Coding. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems, Date-Modified = 2010-10-29 01:40:00 +0200*, pages 43–67. Springer US, 2010. WOS - ISBN: 978-1-4419-6344-4.

[56] R.K. Megalingam, K.B. Venkat, S.V. Vineeth, M. Mithun, and R. Srikumar. Hardware Implementation of Low Power, High Speed DCT/IDCT Based Digital Image Watermarking. In *International Conference on Computer Technology and Development (ICCTD)*, volume 1, pages 535 –539, nov. 2009.

[57] Mentor Graphics. Catapult c. 2010.

[58] G. Moretti. System-level design merits a closer look: the complexity of today's designs requires system-level design, but EDA-tools development is lagging behind the needs of semiconductor and system companies. EDA tools must support system-level design. (design feature). *Electronics Design Strategy, News (EDN)*, 2002.

[59] T. Parr. A functional language for generating structured text. *URL http://www. cs. usfca. edu/˜ parrt/papers/ST. pdf*, 2006.

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

[61] F. Plavec, Z. Vranesic, and S. Brown. Towards compilation of streaming programs into fpga hardware. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 67 –72, sept. 2008.

[62] Jacques Poncin. Utilisation de la transformation de hadamard pour le codage et la compression de signaux d'images. In *Springer-Annals of telecommunications*, pages 235–252, 1971.

[63] The SAC project home page. http://www.sac-home.org.

[64] R. F. Rice. Some practical universal noiseless coding techniques. *Technical Report 79-22*, 1979.

[65] Gupta R.K. and De Micheli G. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design and Test of Computers*, 10:29–41, 1993.

[66] G. Roquier, M. Wipliez, M. Raulet, J.W. Janneck, I.D. Miller, and D.B. Parlour. Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In *IEEE workshop on Signal Processing Systems (SiPS)*, pages 281–286, 2008.

[67] Ghislain Roquier, Matthieu Wipliez, Mickaël Raulet, Jörn W. Janneck, Ian D. Miller, and David B. Parlour. Automatic software synthesis of dataflow program: An MPEG-4 Simple Profile decoder case study. In *Proceedings of 2008 IEEE Workshop on Signal Processing Systems (SiPS 2008)*, pages 281–286, Washington, DC, USA, 2008. IEEE.

[68] W.W. Royce. Managing the Development of Large Software Systems. In *IEEE proceedings*, pages 1–9. IEEE WESCON, 1970.

[69] P.R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010.

[70] Clifford A Shaffer and Hanan Samet. Optimal quadtree construction algorithms. *Computer Vision, Graphics, and Image Processing*, 37(3):402 – 419, 1987.

[71] SIA. Global sales report 2011: http://www.sia-online.org/news/2012/02/06/global-sales-report-2012/semiconductor-industry-posts-record-breaking-revenues-despite-2011-challenges/. February 2012.

[72] N. Siret, M. Wipliez, J. 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, oct. 2010.

[73] P. Strobach. Tree-structured scene adaptive coder. *IEEE Trans. commun.*, 38(4):477 – 486, 1990.

[74] W.R Sutherland. On-Line Graphical Specification Of Computer Procedures. Technical report, MIT, 1966.

[75] Synopsys. Synphony c compiler.

[76] Michael A. Webster, Karen K. De Valois, and Eugene Switkes. Orientation and spatial-frequency discrimination for luminance and chromatic gratings. *J. Opt. Soc. Am. A*, 7(6):1034–1049, Jun 1990.

[77] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

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

[79] M. Wipliez, G. Roquier, M Raulet, J.F. Nezan, and O. Déforges. Code generation for the MPEG Reconfigurable Video Coding framework: From CAL actions to C functions. In *IEEE International Conference on Multimedia and Expo (ICME)*, pages 1049–1052, 2008.

[80] Matthieu Wipliez. *Compilation Infrastructure of Dataflow Programs*. PhD thesis, IETR, INSA Rennes, 35043 Rennes, France, December 2010.

[81] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. Software code generation for the RVC-CAL language. *Journal of Signal Processing Systems*, 63(2):203–213, May 2011.

[82] Hervé Yviquel, Jani Boutellier, Mickaël Raulet, and Emmanuel Casseau. Automated design of networks of Transport-Triggered Architecture processors using Dynamic Dataflow Programs. *Signal Processing Image Communication, Special issue on Reconfigurable Video Coding*, 2013.