System-Level Hardwa Synthesis of Dataflow Programs with HEVC as Study Use Case

Mariem Abid

Thèse

UNIVERSITE BRETAGNE LOIRE

THESE INSA Rennes
sous le sceau de l'Université Bretagne Loire
pour obtenir le titre de
DOCTEUR DE L'INSA RENNES
Spécialité : Traitement du Signal et de l`image

présentée par
Mariem Abid
ECOLE DOCTORALE : *MATISSE*
LABORATOIRE : *IETR*

System-Level Hardware Synthesis of dataflow programs with HEVC as study use case

Thèse soutenue le 28.04.2016
devant le jury composé de :

Mohamed Akil
Professeur à l'ESIEE Paris (France) / Président et Rapporteur
Ahmed Chiheb Ammari
Associate Professor à Université du roi Abdulaziz à Jeddah (Arabie Saoudite) / Rapporteur
Mohamed Atri
Maître de Conférences à Faculté des Sciences de Monastir (Tunisie) / Examinateur
Audrey Queudet
Maître de conférences à l'université de Nantes (France) / Examinateur
Olivier Déforges
Professeur à l'INSA de Rennes (France) / Directeur de thèse
Mohamed Abid
Professeur à l'Ecole Nationale d'Ingénieur de Sfax (Tunisie) / Directeur de thèse

System-Level Hardware Synthesis of Dataflow Programs with HEVC as Study Use Case

Mariem Abid

En partenariat avec

Dedication

To the loving memory of my mother,
Taicir,
who has made me the person I am becoming.

To my father,
Najib,
who has been my role-model for hard work.

To my sister,
Mona,
who has been my emotional anchor.

To my husband,
Hamza,
who has shared the many sacrifices for completing this dissertation.

To my son,
Elyes,
who is my lucky charm.

Abstract

Image and video processing applications are characterized by the processing of a huge amount of data. The design of such complex applications with traditional design methodologies at low-level of abstraction causes increasing development costs. In order to resolve the above mentioned challenges, Electronic System Level (ESL) synthesis or High-Level Synthesis (HLS) tools were proposed. The basic premise is to model the behavior of the entire system using high-level specifications, and to enable the automatic synthesis to low-level specifications for efficient implementation in Field-Programmable Gate Array (FPGA). However, the main downside of the HLS tools is the lack of the entire system consideration, i.e. the establishment of the communications between these components to achieve the system-level is not yet considered.
The purpose of this thesis is to raise the level of abstraction in the design of embedded systems to the system-level. A novel design flow was proposed that enables an efficient hardware implementation of video processing applications described using a Domain Specific Language (DSL) for dataflow programming. The design flow combines a dataflow compiler for generating *C*-based HLS descriptions from a dataflow description and a *C*-to-gate synthesizer for generating Register-Transfer Level (RTL) descriptions. The challenge of implementing the communication channels of dataflow programs relying on Model of Computation (MoC) in FPGA is the minimization of the communication overhead. In this issue, we introduced a new interface synthesis approach that maps the large amounts of data that multimedia and image processing applications process, to shared memories on the FPGA. This leads to a tremendous decrease in the latency and an increase in the throughput. These results were demonstrated upon the hardware synthesis of the emerging High-Efficiency Video Coding (HEVC) standard.

Résumé

Les applications de traitement d'image et vidéo sont caractrisées par le traitement d'une grande quantité de données. La conception de ces applications complexes avec des méthodologies de conception traditionnelles bas niveau provoque l'augmentation des coûts de développement. Afin de résoudre ces défis, des outils de synthèse haut niveau ont été proposés. Le principe de base est de modéliser le comportement de l'ensemble du système en utilisant des spécifications haut niveau afin de permettre la synthèse automatique vers des spécifications bas niveau pour implémentation efficace en FPGA. Cependant, l'inconvénient principal de ces outils de synthèse haut niveau est le manque de prise en compte de la totalité du système, c.-à-d. la création de la communication entre les différents composants pour atteindre le niveau système n'est pas considérée.
Le but de cette thèse est d'élever le niveau d'abstraction dans la conception des systèmes embarqués au niveau système. Nous proposons un flot de conception qui permet une synthèse matérielle efficace des applications de traitement vidéo décrites en utilisant un langage spécifique à un domaine pour la programmation flot-de-données. Le flot de conception combine un compilateur flot-de-données pour générer des descriptions à base de code *C* et d'un synthétiseur pour générer des descriptions niveau de transfert de registre. Le défi majeur de l'implémentation en FPGA des canaux de communication des programmes flot-de-données basés sur un modèle de calcul est la minimisation des frais généraux de la communication. Pour celà, nous avons introduit une nouvelle approche de synthèse de l'interface qui mappe les grandes quantités des données vidéo, à travers des mémoires partagées sur FPGA. Ce qui conduit à une diminution considérable de la latence et une augmentation du débit. Ces résultats ont été démontrés sur la synthèse matérielle du standard vidéo émergent High-Efficiency Video Coding (HEVC).

Acknowledgment

“ You gave me your time, the most thoughtful gift of all”
Dan Zadra

Firstly, I would like to express my sincere gratitude to my thesis directors Prof. Mohamed Abid and Prof. Olivier Déforges. My special recognition goes out to Prof. Mohamed Abid who gave me the opportunity to join the ENIS-CES Lab and offered me the wonderful research opportunity to join INSA-IETR Lab through
a cotutelle agreement. My sincere appreciation and gratitude to Prof. Olivier Déforges for the continuous support of my Ph.D study, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. Thank you very much to both of you for your human exchange and for understanding and considering my personal circumstances

Besides, I would like to thank my advisor Dr. Mickael Raulet, for his guidance, for all the invested time in the supervision of my thesis and for his readiness for listening to the little problems and roadblocks that unavoidably crop up in the course of performing research. In addition, I acknowledge the very interesting theme he proposed for my thesis and the critical comments on this written work.

Besides my advisors, I would like to thank the jury members: Mr Ahmed Chiheb Ammari, Mr Mohamed Akil, Mr Mohamed Atri and Mrs Audrey Queudet, for honouring my jury, for their precious time reading my thesis defense and for their constructive comments.

I am grateful to my fellow labmates of the image team, Khaled, Hervé, Alexandre, Gildas and Antoine for helping me solving the problems in research. Also I thank my friends in the CES Lab for the scientific discussions and for helping me in the administration issues.

A special thought to Mrs Jocelyne Tremier, Mrs Corinne Calo and Mrs Aurore Gouin for their administrative support in the INSA of Rennes and professionalism

Contents

List of Figures

List of Tables

Listings

Acronyms

ADM	Abstract Decoder Model, pp. v, 36, 42, 45, 55, 118
AI	All Intra, pp. 66, 69
AMVP	Advanced Motion Vector Prediction, p. 65
ASIC	Application-Specific Integrated Circuit, pp. v, 1, 7, 8, 91, 115
AVC	Advanced Video Coding, pp. 1, 9, 54, 63, 66, 115
BDTI	Berkeley Design Technology Inc., p. 20
BRAM	block RAM, pp. 8, 68, 72, 88, 108
BSD	Bitstream Syntax Description, p. 36
BSD	Berkeley Software Distribution, pp. 42, 68
BSDL	Bitstream Syntax Description Language, p. 36
BSV	Bluespec SystemVerilog, p. 23
CABAC	Context-Adaptive Binary Arithmetic Coding, p. 65
CAD	Computer Aided Design, pp. 15, 24
CAL	Caltrop Actor Language, pp. v–vii, ix–xiii, xvi, 2, 3, 23, 24, 33, 35–45, 53–55, 57, 58, 62, 63, 66, 68–73, 77–79, 84–86, 89, 90, 92, 93, 97–99, 103, 106, 107, 116–121, 124, 125
CU	Coding Block, pp. 63, 65
CDFG	Control Data Flow Graph, p. 18
CE	Chip-Enable, p. 82
CP	Critical Path, pp. 68, 70
CSDF	Cyclo-Static Dataflow, p. 40
CSP	Communicating Sequential Process, p. 23
CTB	Coding Tree Block, p. 63
CTU	Coding Tree Unit, p. 63
CU	Coding Unit, p. 63
DBF	Deblocking Filter, pp. 65, 66, 86–88, 124
DPB	Decoding Picture Buffer, pp. 65, 66, 69, 88, 89, 98
DPN	Dataflow Process Network, pp. x, 2, 3, 23, 33, 40–42, 45, 55, 58, 60, 62, 77, 79, 80, 97, 116–119, 121, 123
DSE	Design-Space Exploration, pp. vi, ix, 2, 3, 19, 20, 24, 55, 71, 72, 97, 98, 116, 117
DSL	Domain Specific Language, pp. i, 2, 23, 24, 33, 37, 57, 116, 119
DSP	Digital Signal Processing, pp. 20, 23, 68
DSP	Digital Signal Processor, pp. 1, 115
EDA	Electronic Design Automation, pp. ix, 11, 14, 15, 24
ESL	Electronic System Level, pp. i, 18, 120
FF	Flip-Flop, pp. ix, 8, 68
FIFO	First-In First-Out, pp. x, xii, xiii, 19, 23, 39, 40, 42, 43, 53, 58–62, 69, 70, 77, 79–82, 85, 86, 93, 97, 119, 121, 123, 124
FND	FU Network Description, p. 36
FNL	FU Network Language, pp. ix, x, 35, 36, 66, 118, 124

FPGA	Field-Programmable Gate Array, pp. i, v, ix, 1, 2, 7–9, 17, 20, 24, 33, 43, 55, 57, 62, 68, 69, 73, 77, 82, 85, 86, 92, 97, 103, 115, 116, 121, 124, 125
Fps	Frames per Second, pp. 68–70, 72, 85, 86, 90, 92, 111
FSM	Finite-State Machine, pp. x, xiii, 23, 37, 39, 40, 54, 60, 77, 78, 82, 120
FU	Functional Unit, pp. vi, ix–xii, xv, 35–37, 39, 66, 69–72, 85–89, 91, 92, 98, 118, 119, 124
GOP	Group Of Pictures, p. 66
GPP	General-Purpose Processor, pp. 1, 115
GUI	Graphical User Interface, p. 20
HD	High Definition, pp. 1, 9, 69, 115
HDL	Hardware Description Language, pp. v, vii, 3, 15–17, 21, 24, 42–45, 68, 72, 83, 91, 98, 107, 117, 120, 121
HEVC	High-Efficiency Video Coding, pp. i, vi, vii, ix–xii, xv, 1, 3, 9, 53–55, 62–72, 77, 85–87, 89–93, 97–99, 111, 115, 117, 124, 125
HLS	High-Level Synthesis, pp. i, v–vii, ix–xii, 2, 3, 17–22, 24, 55, 57–62, 68, 69, 71–73, 77–79, 81, 82, 85, 86, 88, 89, 91, 92, 97, 98, 103, 106–108, 113, 117, 120, 121, 123–125
HM	HEVC Test Model, p. 66
I/O	Input/Output, pp. 8, 53
IC	Integrated Circuit, pp. 2, 7, 8, 15, 17, 19, 21, 24
IDCT	Inverse Discrete Cosine Transform, p. 65
IDE	Integrated Development Environment, pp. 43, 68
IETR	Institute of Electronics and Telecommunications of Rennes, pp. 53, 99
IR	Intermediate Representation, pp. 42–44, 53, 54, 121
ISO/IEC	International Standardization Organization/International Electrotechnical Commission, pp. 9, 35, 36, 62, 118
ITRS	International Technology Roadmap for Semiconductors, pp. 2, 11, 24, 115, 116
ITU–T	International Telecommunication Union–Telecommunication sector, pp. 9, 62
JADE	Just-in-time Adaptive Decoder Engine, p. 44
JCT–VC	Joint Collaborative Team on Video Coding, pp. 62, 66, 124
KPN	Kahn Process Network, pp. 23, 40, 42, 118, 119
LD	Low Delay, pp. 66, 68
LLVM	Low Level Virtual Machine, p. 44
LSI	Large Scale Integration, p. 15
LUT	Look-up table, pp. ix, 8, 68–70, 72, 86
M2M	Multi-to-Mono, pp. ix, 54, 59, 77, 82
MC	Motion Compensation, pp. 63, 65, 66
MD5	Message Digest 5, p. 66
MoC	Model of Computation, pp. i, vi, ix, 2, 22–24, 36, 39–42, 45, 55, 58, 60, 73, 77, 79, 93, 98, 116, 118, 119
MPEG	Moving Picture Experts Group, pp. vii, x, 1–3, 9, 24, 35–37, 40, 45, 62, 63, 66, 92, 115–118
MPSoC	multiprocessor System On Chip, pp. 1, 7, 97, 115
ms	milliseconds, pp. 68, 70, 72, 86, 89–92
MSI	Medium Scale Integration, p. 15
MV	Motion Vector, pp. 65, 66
NAL	Network Abstraction Layer, p. 65
OpenDF	Open Dataflow environment, pp. vi, 42, 43, 53, 92, 120
Orc–apps	Open RVC-CAL Applications, pp. 66, 85

Orcc	Open RVC-CAL Compiler, pp. vi, vii, ix, x, xiii, 2, 42–45, 53–55, 58, 59, 62, 68, 69, 71, 77, 83–85, 91, 92, 97, 98, 103, 106–108, 117, 120, 121, 125
OS	Operating System, p. 57
PB	Prediction Block, pp. 63, 65
P&R	place and route, pp. 15, 18, 22, 68
PSDF	Parameterized Synchronous Dataflow, p. 40
PU	Prediction Unit, pp. 63, 65
QCIF	Quarter Common Interface Format, pp. 1, 115
QP	Quantization Parameter, pp. 65, 66, 68, 69, 111
RA	Random Access, pp. 66, 69
RAM	Random-Access Memory, pp. x, xiii, xvi, 8, 68–70, 72, 82–86, 93, 97, 98, 108, 123, 125
ROM	Read-Only Memory, p. 8
RTL	Register-Transfer Level, pp. i, v, ix, x, 2, 3, 12, 14–22, 43, 55, 60–62, 68, 71, 73, 82, 84, 91, 97, 103, 116, 117, 120, 121, 123
RVC	Reconfigurable Video Coding, pp. v–vii, ix–xiii, xvi, 2, 3, 24, 33, 35–45, 53–55, 57, 58, 62, 63, 66, 68–73, 77–79, 84–86, 89, 90, 92, 93, 97–99, 103, 106, 107, 116–121, 124, 125
SAO	Sampling Adaptative offset, pp. 65, 66, 86–88, 91, 124
SDF	Synchronous Dataflow, pp. 23, 40, 118
SDRAM	Synchronous Dynamic RAM, p. 69
SIA	Semiconductor Industry Association, p. 11
SOC	System On Chip, pp. 7, 21
SP	Simple Profile, pp. x, 92
SPICE	Simulation Program for Integrated Circuit Emphasis, p. 15
Sps	Samples per Second, pp. 68–70, 85, 86, 89–91
SSA	Static Single Assignment, pp. 42, 53
SSI	Small Scale Integration, p. 15
TB	Transform Block, p. 63
TCE	Transport-Trigger Architecture (TTA)-based Co-design Environment, p. 44
Tcl	Tool Command Language, pp. 20, 71, 72
TTA	Transport-Trigger Architecture, pp. xvii, 44
TU	Transform Unit, pp. 63, 65, 66
UHD	Ultra-High Definition, pp. 1, 98, 115
ULSI	Ultra Large Scale Integration, pp. 15, 17
URQ	Uniform-Reconstruction Quantization, p. 65
VCEG	Video Coding Experts Group, pp. 9, 62
VHDL	VHSIC Hardware Description Language, pp. vii, x, xiii, 16–18, 20, 23, 43, 44, 53, 55, 60–62, 83–85, 91, 103, 108–110, 120
VHSIC	Very High-Speed Integrated Circuits, pp. vii, xvii, 16
VLSI	Very Large Scale Integration, p. 15
VTL	Video Tool Library, pp. 36, 118
WE	Write-Enable, p. 82
WPP	Wavefront Parallel Processing, p. 63
XDF	XML Dataflow Format, pp. x, xiii, 35, 36, 44, 55, 107
XLIM	XML Language-Independent Model, pp. ix, 42–44, 53, 54, 120, 121
XML	eXtensible Markup Language, pp. x, xvii, 35, 42, 53, 62

1 Introduction

> *If we knew what it was we were doing, it would not be called research, would it?*
>
> Albert Einstein

1.1 Context and Motivation

This thesis presents a methodology for implementing video de-compression algorithms using FPGA. The design of such complex systems is becoming extremely challenging due to several factors.

Video compression algorithms are increasingly complex. Video compression is the core technology used in multimedia-based consumer electronics products (i.e., *embedded multimedia systems*) such as digital and cell-phone cameras, video surveillance systems and so on. Over the years, the MPEG video coding standards have evolved from MPEG-1 to MPEG-4/Advanced Video Coding (AVC) to HEVC. Moreover, video resolutions have increased from Quarter Common Interface Format (QCIF) (144p) to High Definition (HD) (1080p) to Ultra-High Definition (UHD) (4K and 8K), resulting in $\sim 1000\times$ increase in resolution complexity relative to QCIF. Besides higher resolutions, the key reason behind increasing video coding complexity is the complex tool set of advanced video encoders. For example, unlike previous standards, the state-of-the-art video coding standard HEVC adopts highly advanced encoding techniques to achieve high compression efficiency for HD and UHD resolution videos at the cost of additional computational complexity ($\sim 3\times$ relative to H.264).

Embedded systems design process has become remarkably difficult. Designing embedded systems involves mapping the target application onto a given implementation architecture. However, these systems have stringent requirements regarding size, performance, real-time constraints, time-to-market and energy consumption. Therefore, meeting up these tight requirements is a challenging task and requires new automation methodologies and ever more efficient computational platforms. There are different possible hardware implementation platforms ranging from processor based embedded systems (such as General-Purpose Processors (GPPs), Digital Signal Processors (DSPs), multiprocessor System On Chips (MPSoCs), etc.) to FPGAs and ASICs. The selection of an appropriate hardware depends upon the applications requirements. However, in order to handle computationally intensive, data-intensive and real-time video compression applications, there is increasing need for very high computational

power. So, what kind of hardware platform is best suited for the real-time application under consideration? In contrast to processor based embedded systems, hardware implementations based on FPGAs and ASICs have proved to be the right choice due to their massively parallel processing exploitation which results in high speed processing.

Embedded system design faces a serious productivity gap. According to the International Technology Roadmap for Semiconductors (ITRS), improvements in the design productivity is not keeping pace with the improvements in semiconductor productivity. This gives rise to an exponentially increasing "design productivity gap" –the difference between the growth rate of Integrated Circuits (ICs) complexity measured in terms of the number of logic gates or transistors per chip and the growth rate of designer productivity offered by design methodologies and tools.

Reference methodologies are no longer suitable. The traditional ways of providing MPEG video coding specifications based on textual descriptions and on $C/C++$ monolithic reference software specifications are becoming not suitable for parallel architectures. On the one hand, such specification formalism do not enable designers to exploit the clear commonalities between the different video CODECs, neither at the level of specification nor at the level of implementation. On the other hand, mapping the $C/C++$ monolithic reference software onto parallel architectures, such as FPGAs, means rewriting the source code completely in order to distribute the computations on the different processing units, which is a tedious and time-consuming task. In order to improve the re-use and time-to-market, there is a great need to develop design and verification methodologies that will accelerate the current design process so that the design productivity gap can be narrowed.

1.2 Problem Statement and Contributions

Many questions arise about suitable approaches to bridge the design productivity gap as well as the gap between traditional sequential specifications and final parallel implementations. On the one hand, according to the ITRS, enhancement in design productivity can be achieved by increasing the level of abstraction beyond RTL and by employing design reuse strategies. On the other hand, system-level design has emerged as a novel design methodology to fill the gap between specification and implementation in traditional methodologies. Raising abstraction level to system-level allows the designer to handle the complexity of the entire system disregarding low-level implementation details and thus results in fewer numbers of components to handle. However, the key challenge in raising the level of abstraction to system-level is to deal with system integration complexity and perform DSE, means that *developers need to know how to pull together the different components through efficient communication mechanisms while allowing for system-level optimizations*. Moreover, in system-level design, verification is critical in the design process, which enables to assert that the system meets its intended requirements.
Within this context, and knowing the drawbacks of the past monolithic specification of video standard, efforts have focused on standardizing a library of video coding components called RVC. The key concept behind the standard is to be able to design a decoder at a higher level of abstraction than the one provided by current monolithic C-based specifications, while ensuring parallelism exploitation, modularity, reusability and reconfigurability. The RVC framework is built upon a dataflow-based Domain Specific Language (DSL) known as RVC-CAL, a subset of CAL. RVC is based on dynamic dataflow programming. The MoC used to specify the way data is transferred and processed is known as DPN. The objective of this thesis is then to propose a new rapid prototyping methodology of DPN-based dataflow programs on FPGAs. Several issues could be raised namely how to translate the DPN-based programs into RTL descriptions suitable for implementation in programmable hardware, while reducing the complexity and time-to-market, and obtaining performance efficient implementation. Several works have sought to address these issues, but provided only partial solutions for synthesis at the system-level.
Motivated by these developments, our contributions with respect to the challenges of implementing dynamic dataflow programs onto FPGAs are as follows.

- First, we propose a novel automated design flow for rapid prototyping of RVC-based video decoders, whereby a system-level design specified in RVC-CAL dataflow language is quickly

translated to a hardware implementation. Indeed, we design image de-compression algorithms using the actor oriented language under the RVC standard. Once the design is achieved, we use a dataflow compilation infrastructure called Orcc to generate a *C*-based code. Afterward, a Xilinx HLS tool called Vivado is used for an automatic generation of synthesizable hardware implementation.

- Then, we propose a new interface synthesis method that enables the enhancement of the implementation of the communication channels between components and therefore the enhancement of scheduling policies, aiming at optimizing performance metrics such as latency and throughput of dataflow-based video decoders. Therefore, a new system level implementation is elaborated based on this optimized implementation of the communication and scheduling mechanisms.

- Next, we investigate techniques as an aid for DSE for achieving high performance implementations by exploiting task or data-level parallelism at the system-level.

- Finally, we present a framework for system or component-level verification. Hence, we have demonstrated the effectiveness of our proposed rapid prototyping by applying it to an RVC-CAL implementation of the HEVC decoder, which is very challenging because it typically involves high computational complexity and massive amounts of data processing.

1.3 Outline

This thesis is structured as follows.
Part I outlines the background to the study including its theoretical framework. Following the thesis's introduction, Chapter 2 entails an overview of trends and challenges in embedded systems design and the emergence of system-level design of embedded systems. Chapter 3 reviews basic properties of the dataflow programming, introduces the MPEG-RVC framework as well as its reference programming language and the semantics of the DPN model. Then, it summarizes the existing HDL code generation approaches from dataflow representations.
Part II presents the major contributions of this thesis. In Chapter 4, a rapid prototyping methodology for DPN-based programs is presented. The proposed design flow combines a dataflow compiler for generating *C*-based HLS descriptions from a dataflow description and a *C*-to-gate synthesizer for generating RTL descriptions. The results obtained applying to an RVC-CAL HEVC decoder are discussed. Chapter 5 presents optimization techniques by proposing new interface synthesis method firstly and further by exploiting all the features of dynamic dataflow. Chapter 6 concludes the two parts of this thesis and discusses directions for future work.
Part III provides supplementary information to this thesis. In Appendix A, we present a user guide to the hardware generation from dataflow programs using our proposed design flow. Appendices B and C present all available HEVC test sequences and a summary of Vivado HLS directives, respectively. Finally, Appendix D provides a brief explanation of the work presented in this thesis in French.

1.4 Publications

The work presented in this thesis is partly published in the following publications.

International Journal paper

M. Abid, K. Jerbi, M. Raulet, O. Deforges, and M. Abid. Efficient system-level hardware synthesis of dataflow programs using shared memory based FIFO: HEVC decoder case study. submitted to the Journal of Signal Processing Systems (under review), 2015.

International Conference paper

M. Abid, K. Jerbi, M. Raulet, O. Deforges, and M. Abid. System level synthesis of dataflow programs: HEVC decoder case study. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pages 16, May 2013.

Part I

BACKGROUND

2

Fundamentals of Embedded Systems Design

> *Basic research is what I'm doing when I don't know what I'm doing.*
>
> Wernher von Braun

2.1 Introduction

In this chapter, I will look at the emergence of system-level design of embedded systems whose functionality involves real-time processing of media streams [Neuendorffer and Vissers, 2008], e.g., streams containing video data. In order to justify the needs of system-level design and the driving force behind its emergence, I must first set the stage for my study of embedded systems design and highlight challenges and complexities involved in designing embedded systems in Section 3.2. In section 2.3, I will give an overview of existing design methodologies linked to various levels of abstraction at which design process can be approached. Moreover, a taxonomy of design automation is developed by using the Y-chart [Gajski and Kuhn, 1983] as a reference point. Section 2.4 and section 2.5 review the traditional design flows for embedded system design, including a discussion of their limitations. Section 2.6 presents motivating trends towards using system-level design.

2.2 The Embedded Systems Design

This section discusses some general aspects of embedded system design and highlights challenges and complexities involved in designing embedded systems.

2.2.1 What is an embedded system?

Embedded systems are defined as information processing systems embedded into enclosing products such as cars, telecommunication or fabrication equipment. Such systems come with a large number of common characteristics, including real-time constraints, and dependability as well as efficiency requirements [Marwedel, 2006]. A particular class of embedded systems are real-time systems. A real-time system is one in which the response to an event must occur within a specific time, otherwise the system is considered to have failed [Dougherty and Laplante, 1995]. The importance of embedded systems is growing continuously. That is, the evolution of embedded systems parallels Moore's Law [Moore, 1965] which states that the number of transistors on Integrated Circuits (ICs) doubles approximately every two years. This IC technology advance enabled the design of embedded systems on a single chip called System On Chip (SOC). A

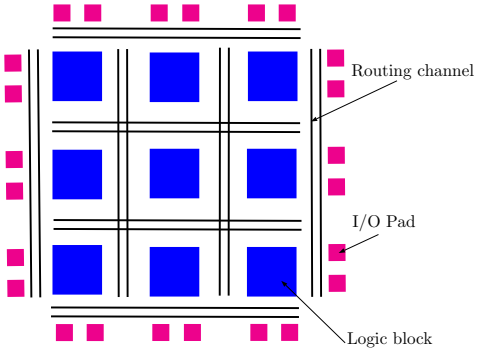


Figure 2.1: The basic FPGA structure: a logic block consists of a 4-input LUT, and a FF.

SOC is defined as a device which is designed and fabricated for a specific purpose, for exclusive use by a specific owner [Amos et al., 2011]. In other words, a SOC is a single piece of silicon that contains all circuits required to deliver a set of functions. It may include on-chip memory, embedded processor, peripheral interfaces, and other components necessary to achieve the intended function. It may comprise more than one processor core, referred to as multiprocessor System On Chip (MPSoC), where each of the embedded core whill take care of different sub-functions. SOCs can be implemented as Application-Specific Integrated Circuits (ASICs) or using Field-Programmable Gate Arrays (FPGAs).

2.2.1.1 What is an ASIC?

An ASIC is a unique type of IC meant for a specific application. Developing an ASIC takes very much time and is expensive. Furthermore, it is not possible to correct errors after fabrication.

2.2.1.2 What is a FPGA?

An FPGA is a reprogrammable IC, i.e it can be programmed for different algorithms after fabrication. FPGA addresses the cost issues inherent in ASIC fabrication.

FPGA architecture [1] The basic structure of an FPGA is composed of the following elements:

- Look-up table (LUT): this element performs logic operations.
- Flip-Flop (FF): this register element stores the result of the LUT.
- Wires: these elements connect resources to one another.
- Input/Output (I/O) pads: these physically available ports get data in and out of the FPGA.

The combination of these elements results in the basic FPGA architecture shown in Figure 2.1. The FPGA fabric includes embedded memory elements that can be used as Random-Access Memory (RAM), Read-Only Memory (ROM), or shift registers. These elements are block RAMs (BRAMs), LUTs, and shift registers.

- The BRAM is a dual-port RAM module instantiated into the FPGA fabric to provide on-chip storage for a relatively large set of data. The two types of BRAM memories available in a device can hold either 18 k or 36 k bits. The number of these memories available is device specific. The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations.
- The LUT is a small memory in which the contents of a truth table are written during device configuration.
- The shift register is a chain of registers connected to each other. The purpose of this structure is to provide data reuse along a computational path.

[1] http://www.xilinx.com

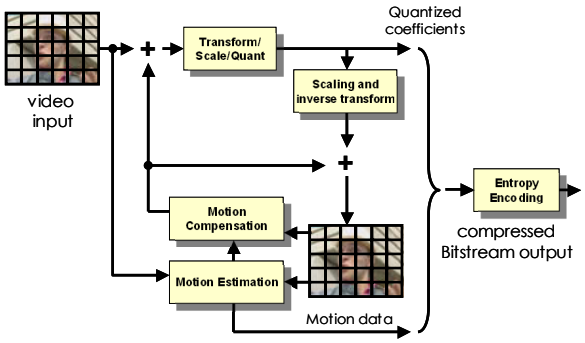


Figure 2.2: Hybrid video encoder [Jacobs and Probell, 2007].

Commercial FPGA devices Since the release of the first Xilinx XC2064 commercial FPGA in 1985, the market of FPGAs has been steadily growing. Xilinx and Altera are the two main FPGA manufacturers. The latest FPGAs are Xilinx Virtex-7 and Altera Stratix-V which are targeting highest performance and capacity. For instance, Virtex-7 FPGA delivers up to 2 million programmable logic cells (LUTs) and offer more than 4Tbps of serial bandwidth.

FPGA parallelism Unlike processors, FPGAs are inherently parallel, so that different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a specific section of the circuit, and can run independently without depending on any other logic blocks. As a result, the performance of one part of the application is not affected when additional processing is added. The parallel computing power of FPGA is well suited for algorithms requiring high bandwidth and for the calculation of many operations in parallel on video data such as real-time video processing algorithms.

2.2.2 Video compression in FPGA

For a video processing perspective, read-time can mean that the total processing per pixel must be completed within a pixel sample time [Bailey, 2011]. Real-time video processing need in embedded systems arises for video telephony, digital cameras, digital television, high definition TV decoders, DVD players, video conferencing, internet video streaming and other systems. All video storage and transmission mechanisms rely heavily on video compression and decompression systems (known as CODECs). To enable interoperability, most products use standards-based digital video compression techniques. Video coding standards have evolved through the development of the International Standardization Organization/International Electrotechnical Commission (ISO/IEC) and International Telecommunication Union–Telecommunication sector (ITU–T) standards. The ISO/IEC Moving Picture Experts Group (MPEG)–1 [ISO/IEC 11172-2:1993], MPEG–4 Visual [ISO/IEC 14496-2:1999], and the ITU–T H.261 [ITU-T Rec. , 12/90], H.263 [ITU-T Rec. H.263 , 03/96] are some of the popular international video compression standards. The essential underlying technology in each of these video compression standards is very similar and uses a hybrid video coding scheme (i.e., motion compensation, transform, quantization, entropy coding) as illustrated in Figure 2.2. The standards differ in the applications they address. Each standard is tuned to perform optimally for a particular application in terms of bit rates and computational requirements: MPEG–1 for CD–ROM, MPEG–4 Visual for Television and Web environments, H.261 for videoconferencing, and H.263 for videophones. The H.264/MPEG–4 Advanced Video Coding (AVC) [ITU-T Rec. H.264 and ISO/IEC 14496-10], jointly developed by ITU–T Video Coding Experts Group (VCEG) and ISO/IEC MPEG, provides up to 50% more bit rate reduction at the same quality of other existing standards. As a result of this significant advancement in compression technology, H.264/ MPEG–4 AVC is used in a wide variety of applications. The growing popularity of High Definition (HD) video, and the emergence of beyond HD formats (e.g., 4k × 2k or 8k × 4k resolution) are creating even stronger needs for coding efficiency superior to H.264/MPEG–4 AVC's capabilities. Need for a codec superior than H.264/MPEG–4 AVC was result in the newest video coding standard High-Efficiency Video Coding (HEVC) [Bross et al., 2012], which was finalized in 2013. HEVC is based on the same structure as prior hybrid video codecs like H.264/MPEG–4 AVC

but with enhancements in each coding stage. The goal of HEVC is to increase the compression efficiency by 50% as compared to that of the H.264/MPEG–4 AVC. The higher compression rate achieved in HEVC results in an increase in computational complexity for video encoding and decoding. HEVC encoders are expected to be several times more complex than H.264/MPEG–4 AVC encoders [Bossen et al., 2012]. There is also a slightly increase in the computational complexity of video decoder. Due to the high computational complexity involved and to the huge amount of data that needs to be processed, high processing power is required to satisfy the real-time constraints. This can be more readily achieved through hardware parallelism. Intuitively, reconfigurable hardware in the form of FPGA has been proposed as a way of obtaining high performance for video processing algorithms, even under real-time requirements. Moreover, implementing video processing algorithms on reconfigurable hardware minimizes the time-to-market cost, enables rapid prototyping of complex algorithms and simplifies debugging and verification. Therefore, FPGAs are an ideal choice for implementation of real-time video processing algorithms.

2.2.3 The embedded systems design challenges

Systems design is the process of deriving, from requirements, a model from which a system can be generated more or less automatically. A model is an abstract representation of a system. For example, software design is the process of deriving a program that can be compiled; hardware design is the process of deriving a hardware description from which a circuit can be synthesized [Henzinger and Sifakis, 2006]. So, why is it so hard to design the real-time embedded system? The design of embedded systems is a challenging issue, for the following reasons:

2.2.3.1 Design constraints

1. *Real-time constraints*: embedded systems have to perform in real-time. If data is not ready by a certain deadline, the system fails to perform correctly. Real-time constraints are *hard*, if their violation causes the failure of the system functioning, and *soft*, otherwise. In the field of embedded video compression systems, there are latency and throughput constraints. The latency constraints states that the interval between the time T_{avail}, when the input data are available to the system, and T_{prod}, when the corresponding output data are produced must be less than the constraint ΔT.

   $$T_{prod} - T_{avail} \leq \Delta T \tag{2.1}$$

   The throughput constraint states that the interval between the time $T^{starting}$, when data processing is starting, and T^{ending}, when data processing is finished must be less than ΔT.

   $$T^{ending} - T^{starting} \leq \Delta T \tag{2.2}$$

2. *Small size and weight*: typically, embedded systems are physically located within some larger device. Therefore, their shape and size may be dictated by the space available and the connections to the mechanical components.

3. *Low Power and low energy consumption*: in mobile applications, embedded designs are powered by batteries. This requires designs with low energy consumption. Power consumption is important also in applications in which the large amounts of heat produced during the circuit operation are difficult to be dispersed.

4. *High-Performance*: an embedded system should perform its functions and complete them quickly and accurately.

5. *Reliability constraints*: embedded systems are often used in life critical applications that is why reliability and safety are major requirements.

6. *Low cost*: embedded systems are very often mass products in highly competitive markets and have to be shipped at a low manufacturing and design cost.

7. *Short time-to-market*: time-to-market is the length of time from the product idea conception until it is available for sale.

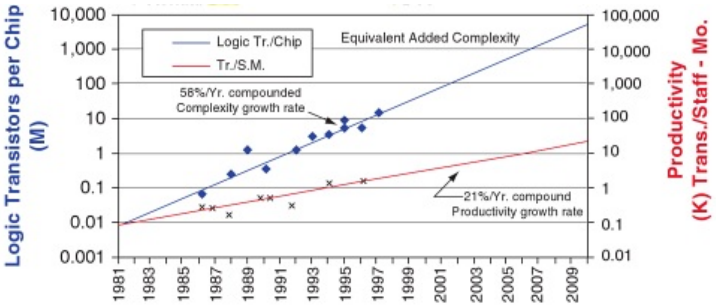


Figure 2.3: Difference between design complexity and design productivity: the productivity gap. Source: Sematech[2]

Embedded Video Codec Design Requirements and Constraints The key challenge for embedded system design is how to implement a system that fulfills a desired functionality (e.g., video compression functionality) and simultaneously optimize the aforementioned design metrics in a timely fashion. First, during design of such real-time embedded systems, ensuring temporal correctness of their behavior is equally important as ensuring its functional correctness. Second, the need to execute complex video processing algorithms under tight timing constraints implies that the embedded systems have to be designed to sustain the ever-increasing computational complexity and to be extremely high performance. Third, to be suitable for the deployment in the consumer electronics products described in Section 2.2.2, these embedded systems must be optimized to have low cost and low power/energy consumption.

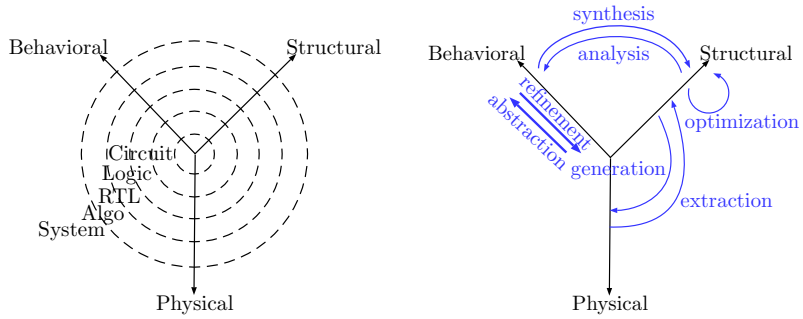
2.2.3.2 Design productivity gap

The Semiconductor Industry Association (SIA)[2] shows that a design productivity gap exists between the available chip capacity and the current design capabilities. Figure 2.3 plots Moore's Law, together with the productivity growth, expressed in transistors per staff member per month over the last decades. Due to improving engineering skills, an increase in the number of transistors that one designer can handle can be observed. The pace at which the design productivity increases is, however, much smaller than the slope of Moore's Low. That is, whereas Moore's Low predicts that the chip capacity doubles every eighteen months, the hardware design productivity in the past few years is estimated to increase at 1.6× over the same period of time. As can be seen from Figure 2.3, the design productivity gap originate from the 1980s. At that moment, it became clear that it was no longer possible in digital design to cope with every transistor individually. This "design crisis" was the driven force behind the introduction of design abstraction levels [Bell and Newell, 1971] together with well-defined design methodologies and the advent of the automation of the design of electronic systems and circuits (Electronic Design Automation (EDA)) [Lavagno et al., 2006]. Hence, design methodologies become a popular research topic to tackle these aforementioned design challenges of embedded systems in the recent decade.

2.3 Hardware Design Methodologies

In order to explain the different design methodologies, I will use the Y-Chart, which was introduced in 1983 by Gajski and Kuhn [Gajski and Kuhn, 1983] and refined by Walker and Thomas [Walker and Thomas, 1985]. The Gajski-Kuhn Y-chart is depicted in Figure 2.4(a). This *model of design representation* is described using three axes, each representing one of three domains of description-behavioral, structural, and physical. The behavioral domain describes the behavior, or functionality, of the design, ideally without any reference to the way this behavior is achieved by an implementation. The structural domain describes the abstract implementation, or logical structure, of the design as a hierarchy of components and their interconnections. The physical domain describes the physical implementation of the design.

[2]Sematech Inc. International Technology Roadmap for Semiconductors (ITRS), 2004 update, design. http://www.itrs.net, 2004.



(a) Levels of abstraction versus domains of description.

(b) Y-tasks.

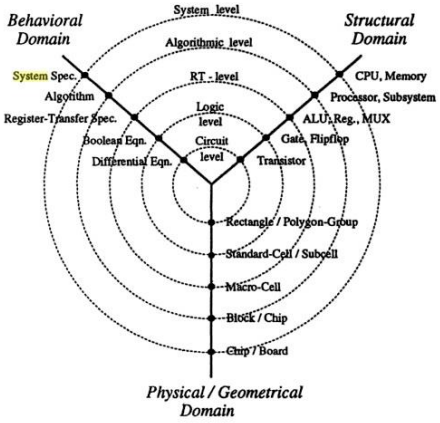
Figure 2.4: Gajski-Kuhn Y-chart.

2.3.1 Levels of abstraction

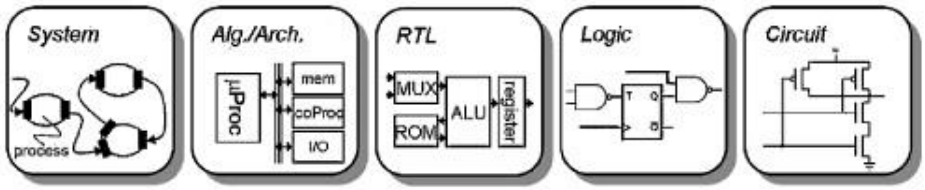
The concentric circles of the Y-chart represent the different levels of abstraction (Figure 2.4(a)). The level of detail increases from the outside inward.

- System-level: defining the partitions of the system (as processes) and the communication methods and interfaces between the partitions and the outside world. The system-level is concerned with overall system structure and information flow.
- Algorithmic-level, behavioral-level or high-level: behavioral modeling with high-level programming languages of the computation performed by an individual process, i.e., the way it maps sequences of inputs to sequences of outputs.
- Register-Transfer Level (RTL): describing the circuit in terms of registers and the data transfers between them using logical operations (combinational logic).
- Logic-level or gate-level: defining the behavior of RTL components with a set of interconnected logic gates and flip-flops.
- Circuit-level or transistor-level: implementing the behavior of the logic gates with interconnected transistors.

The main components that can be found at the different levels of abstraction are represented graphically in Figure 2.5. As an example, consider a binary counter. At the algorithmic-level, we know that the counter increments its current value, producing a new value. At the next lower level, we understand that to carry out this function, some sort of register is needed to hold the value of the counter. We can state this idea using a register transfer statement such as $AC \leftarrow AC + 1$. On the structural side, the register consists of gates and flip-flops, which themselves consist of transistors [Null and Lobur, 2010].
The Y-chart provides also a convenient framework for the definition of design tasks. They can be expressed as transitions between points on the axes of the chart as illustrated in Figure 2.4(b). Based on these transitions, the terms generation, extraction, synthesis and analysis can be defined. Transitions from the structural domain to the physical domain are called generation, reverse transitions are are called extraction, those from the behavioral to the structural domain are called synthesis, and transitions in the opposite direction are called analysis. The task of synthesis is to take the specifications of the behavior required for a system and a set of constraints and goals to be satisfied and to find a structure that implements the behavior while satisfying the goals and constraints [Mohanty et al., 2008]. The tasks of refinement and optimization can be demonstrated on the Y-Chart as well. Refinement is represented by an arrow on the behavioral axis from a high to a lower abstraction level. On the other hand, optimization can be represented as an arrow at any point in the chart which points back to its starting point. Thus, such optimization is a task that is performed in-place and can occur at any level in any domain. In optimization, the basic functionality remains constant, but the quality of the design (in terms of performance, area and power consumption for example) is improved.
The design methodology is declared as intersections of the domain axes and the abstraction



(a) The main components at different abstractions levels in the Y-chart [Michel et al., 2012].



(b) Graphical representation of the main components at different levels of abstraction [Verhelst and Dehaene, 2009].

Figure 2.5: Different representations of the main components at each abstraction level.

circles in the Y-chart. We will explain in the following some basic system design methodologies related to the different abstraction levels in the Y-chart [Gajski et al., 2009].

2.3.2 Bottom-up methodology

The bottom-up design methodology starts from the lowest abstraction level, and each level generates libraries for the next-higher abstraction level as highlighted in Figure 2.6(a). The advantage of this methodology is that abstraction levels are clearly separated, each with its own library. The disadvantage is that optimal library for a specific design is difficult to achieve since parameters need to be tuned for all the library components at each level.

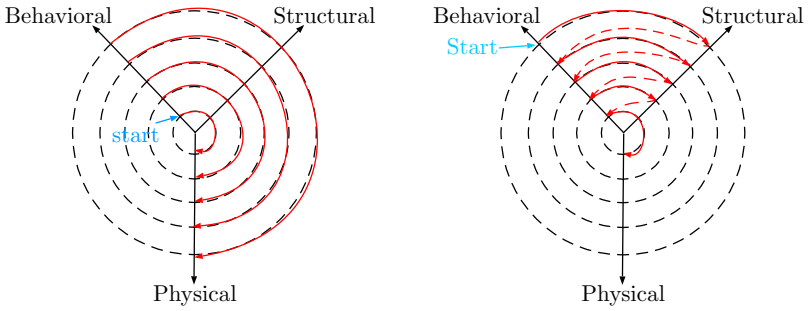
2.3.3 Top-down methodology

In contrast to bottom-up methodology, the top-down methodology starts with the highest abstraction level to convert from functional description of the system into system structure as highlighted in Figure 2.6(b). The advantage of this approach is that high-level customization is relatively easy without implementation details, and only a specific set of transistors and one layout is needed for the whole process. The disadvantage is that it is difficult to get the accurate performance metrics at the high abstraction levels without the layout information.

2.3.4 System design process

The Y-chart *separation of concerns*, i.e. separating application (behavior) from architecture (structure) [Kienhuis et al., 1997], leads to the following five-step approach in the embedded system design process which I detail in the following. In a top-down way, a design always starts from system-level specifications and ends with a physical implementation of the system as depicted in Figure 2.7. The bottom-up process works in reverse.

1. The requirements are the customer's expectations about what the system has to achieve. A system specification includes not only functional requirements-the operations to be per-



(a) The bottom-up methodology. (b) The top-down methodology.

Figure 2.6: Design methodologies in the Y-chart.

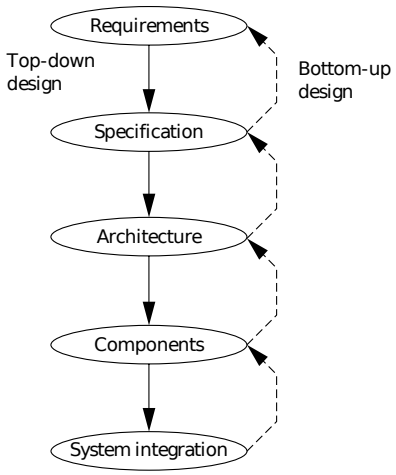


Figure 2.7: Major steps in the embedded systems design process [Wolf, 2008].

formed by the system but also nonfunctional requirements, including speed, power, and manufacturing cost as explained in Section 2.2.3.1.

2. The specification states only what the system does, not how the system does things. Once the specification is determined, the design process involving various levels of abstraction is performed.

3. The architecture gives the system structure in terms of large components.

4. The architectural description tells us what components we need. The component design effort builds those components in conformance to the architecture and specification.

5. After the components are built, the system integration puts together the components to build a complete system.

For the rest of the manuscript, I focus on the top-down design methodology that transform a given high-level system description into a detailed implementation.

2.3.5 Design flow and taxonomy of synthesis

The Y-chart also serves as a *model of design synthesis* using the multiple levels of abstraction and the three domains of description. Design synthesis is a path through the Gajski-Kuhn Y-chart from a high-level (of abstraction) behavioral domain description to a low-level physical domain description. This design flow includes translating and building inter-domain links from the behavioral to the structural to the physical domain, as well as adding enough additional detail to produce a low-level description from a high-level one. Thus the end goal of design synthesis (or design flow) is to produce a physical domain description at a low enough level to be implemented in hardware [Walker and Thomas, 1985]. Many alternative flows through

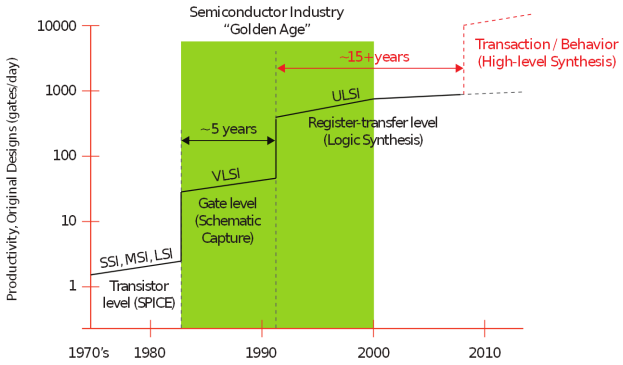


Figure 2.8: The evolution of design methodology adoption in the EDA industry [3].

the Y-chart are possible. We distinguish between transistor-level design, logic-level design, RTL design, high-level design, and system-level design corresponding to the input on circuit-level, logic-level, RTL, algorithmic-level and system-level, respectively.
We expand on all these design flows in the sections that follow, tracing thereby the EDA history [Sangiovanni-Vincentelli, 2003] and the evolution of the design flow over the last decades [Jansen, 2003], aiming at increasing design productivity.

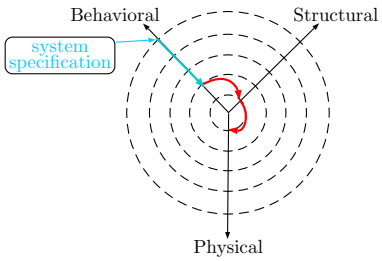
2.4 RTL Design

As shown in Figure 2.8, raising the level of abstraction is one of the major contributors to the design productivity improvement. This chart shows the evolution of IC design from the mid-1970s to the present. There were gradual advancements to the IC technology through Small Scale Integration (SSI), Medium Scale Integration (MSI), Large Scale Integration (LSI), Very Large Scale Integration (VLSI) technology that evolved in the 1970s and the most recent is Ultra Large Scale Integration (ULSI) technology:

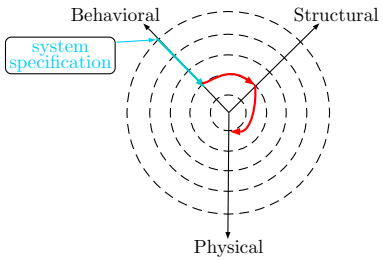
- SSI: contains less than ten logic gates per IC;
- MSI: contains ten to hundred logic gates per IC;
- LSI: contains hundred to ten thousand logic gates per IC;
- VLSI: contains more than ten thousand of logic gates per IC;
- ULSI: contains hundreds of thousands of logic gates per IC.

Since ICs were designed, optimized, and laid out by hand until the late 1960s, new Computer Aided Design (CAD) tools appeared to automate the design process from the 1970s.
From the mid-1970s to the early 1980s, the first incarnation of design synthesis operated at the transistor-level, and is called transistor-level design. At this level, designers used procedural languages to construct and assemble parameterized building blocks. The basic building blocks are transistors, resistors, capacitors, etc. The transistor-level design flow is shown in Figure 2.9(a). The behavioral description is done by a set of differential equations, whereas the physical description of the transistor-level comprises the detailed layout of components and their interconnections. Simulation Program for Integrated Circuit Emphasis (SPICE) is used for the transistor-level simulation to verify whether the logic design specified at the transistor-level will behave the same as the functional specifications. Although SPICE simulation dramatically improved designer productivity of ICs through the 1970s, they are merely verification tools and design automation tools which are needed to speed up the design process and keep up with Moore's Law.

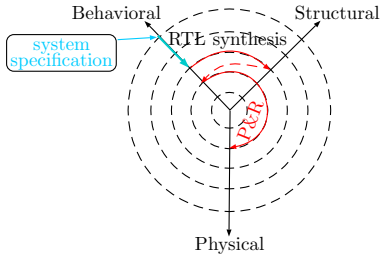
[3] http://www.accellera.org/resources/articles/icdesigntrans



(a) Gasjki-Kuhn Y-Chart for transistor-level design flow.



(b) Gasjki-Kuhn Y-Chart for logic design flow.



(c) Gasjki-Kuhn Y-Chart for RTL design flow.

Figure 2.9: The path of the first, second and third EDA generations in the Y-chart.

By the 1980s, using schematic editors for circuit capture designers were able to build graphs constructed from standard cells-known as *netlists*, and automatically place them on the IC. The logic-level design is shown in Figure 2.9(b). Typical building blocks include simple logic gates, such as *and*, *or*, *xor* and 1-bit 2-to-1 multiplexer, and basic memory elements, such as latch and flip-flop. The behavioral description is done by boolean equations. Automatic place and route (P&R) tools emerged to bridge the gap between the structural and physical domains at the logic-level (rather than at the transistor-level). Moving from the transistor-level design to the logic-level design together with automating the circuit layout were important step toward improving designer productivity. Moreover, by moving the functional verification of the circuit from the transistor-level to the logic-level simulation speed were improved by factors of $100 - 1000\times$ [Stroud et al., 2009; Gries and Keutzer, 2006]. However, designers found that manually entering $30 - 40,000$ gates was simply too time consuming. Worse to verify a system the entire gate-level design had to be entered.
To address this issue, by the early 1990s the logic-level has been abstracted to the RTL which has driven to an over-$100\times$ increase in designer productivity. The RTL synthesis automates the implementation steps below the RTL. When designing at the RTL, designers needed only to describe the logic transfer functions between registers. The detailed logic implementation of those transfer functions need not be described. Hardware Description Languages (HDLs) have been intended to model circuits at the RTL. Based upon these languages, logic synthesis or RTL synthesis tools were developed. Designers were able to automatically transform HDL descriptions of circuits at the RTL into gate-level netlists. The RTL design flow is represented in Figure 2.9(c). In this design flow, the hardware designer would manually refine the behavioral system specifications down to the RTL. From that point, RTL synthesis and P&R complete the design flow.

2.4.1 What is a HDL?

The two principal HDLs currently used include Very High-Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) [IEEE 1076-200] and Verilog [IEEE 1463-2001]. While their syntax is at least reminiscent of high-level software languages, the specification of a circuit in an HDL is different from writing a software program. Software programs have a sequential execution model in which correctness is defined as the execution of each instruction or function in the order it is written. The movement of data is implicit and is left to the underling hardware.

Table 2.1: Truth table of half adder.

A	B	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Memory accesses are inferred, and processors provide implicit support for interfaces to memory. By contrast, hardware designs consist of blocks of circuitry that all run concurrently. Data movement is written explicitly into the model, in the form of wires and ports. Memory and memory accesses must be explicitly declared and handled [Martinez et al., 2008]. Listing 2.1 shows the VHDL implementation of a 1-bit half adder along with its RTL implementation in Figure 2.10. A half adder adds two input bits A and B and generates two outputs a carry and sum. Table 2.1 shows the truth table of a half adder.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 entity halfadder is
5     Port ( A : in  STD_LOGIC;
6            B : in  STD_LOGIC;
7            sum : out  STD_LOGIC;
8            carry : out  STD_LOGIC);
9 end halfadder;
10 architecture Behavioral of halfadder is
11 begin
12 Process (A,B)
13 begin
14    sum <= A XOR B;
15    carry <= A AND B;
16 end process;
17 end Behavioral;
```

Listing 2.1: VHDL code of a 1-bit half adder. Sum and carry are assigned in parallel

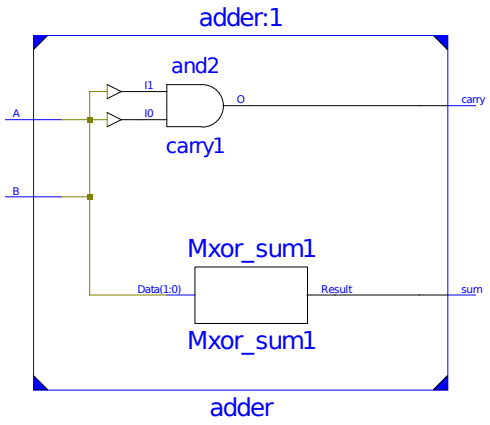


Figure 2.10: RTL schematic of the 1-bit half adder (logic synthesis with Xilinx ISE).

RTL design have several advantages over the traditional schematic-based design [Palnitkar, 2003]:

- Designs can be described at a higher level of abstraction by use of HDLs, without even choosing a specific fabrication technology (technology-independent). If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology.
- By describing designs in HDLs, functional verification can be done early in the design cycle. Since designers work at the RTL, they can optimize and modify the RTL description until it meets the desired functionality.
- Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics, which are almost incomprehensible for very complex designs.

2.4.2 What is wrong with RTL design and HDLs?

There are several issues in RTL design that are simply the result of how HDLs and synthesis tools emerged. Traditional HDLs such as VHDL and Verilog lack many of the high-level and abstraction facilities commonly found in modern mainstream languages such as C++ or Java [Arcas-Abella et al., 2014]. As a consequence, this low-level style of coding requires significant

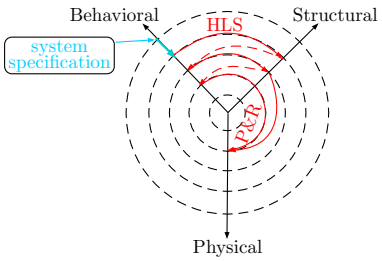


Figure 2.11: Gasjki-Kuhn Y-Chart for HLS Design Flow.

hardware design knowledge and long development cycles. Moreover, low-level synthesis has became tedious and error-prone for non-expert designers. It might also affect expert FPGA designers from the productivity and cost points of view. As shown in Figure 2.8 [4], since about 2005, IC designer productivity has stagnated. Moreover, given that the RTL design has been in use for more than 15 years, it is no longer possible to consider it the forefront design approach that is required to bring us new, exciting consumer and industrial electronic products. That is why, HDLs are constrained to keep up with advances in ULSI technology [Null and Lobur, 2010]. It has become evident that the level of abstraction in IC design must be raised once again to allow designers to think in terms of functions, processes, and systems, as opposed to worrying about gates, signals, and wires. In early 2000s, the move to the next level of abstraction using High-Level Synthesis (HLS) makes it possible to cope with the increasing design complexity and get rid of hand-written HDLs.

2.5 HLS Design

This section presents a survey of the HLS design and highlights its advantages and limitations. HLS, *C* synthesis, Electronic System Level (ESL) synthesis, algorithmic synthesis, or behavioral synthesis improves design productivity by automating the refinement from an algorithmic level specification of the behavior of a digital system to RTL description of the circuit in the form of VHDL or verilog [McFarland et al., 1990a; Coussy and Morawiec, 2008; Martin et al., 2010]. As can be seen on the Y-chart for HLS design flow (Figure 2.11), the transition from the specifications to the start of the automated design flow becomes smaller now compared to the RTL design [Meeus et al., 2012]. From that point, HLS, RTL synthesis and P&R complete the design flow. HLS consists of a sequence of tasks [Andriamisaina et al., 2010]. First, the design specification is written at the algorithmic level by a *C*-based high-level programming language (whether ANSI *C* or *C++*) [Gajski et al., 2010]. At this level the focus is on the computations performed by an individual component and the way it maps sequences of inputs to sequences of outputs. To illustrate, Listing 2.2 shows the *C++* specification of the half adder.

```
1 void HalfAdder(int A, int B, int& Sum, int& Carry)
2 {
3    Sum = A ^ B;
4    Carry = A & B;
5 }
```

Listing 2.2: *C++* code to implement a half adder.

The second step consists in compiling the algorithmic high-level specification into an intermediate representation in the form of various flow graphs such as Control Data Flow Graphs (CDFGs). The following phases consist in operation scheduling, resource allocation and resource binding. During operation scheduling, each operation is scheduled at time steps or clock cycles. Resource allocation determines the type and the number of hardware resources (adders, multipliers, registers) that should be used to implement the design. Then in resource binding, each operation is assigned to the allocated hardware components. Once allocation, scheduling and binding decisions are made, the goal is to generate RTL architecture. Figure 2.12 illustrates the different HLS phases.

[4] http://www.accellera.org/resources/articles/icdesigntrans

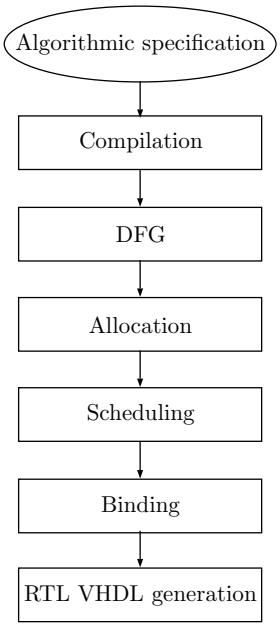


Figure 2.12: Steps of HLS [Andriamisaina et al., 2010].

Several advantages arise from moving the design effort to higher abstraction levels and using HLS in the design flow [McFarland et al., 1990b; Coussy and Takach, 2009].

- Shorter design cycle, which means that a product can be completed faster, decreasing the design cost and the time-to-market.
- Fewer errors and shorter debugging time, since the synthesis process can be verified at the high-level together with the shorter amount of code of high-level specifications.
- Design-Space Exploration (DSE), since HLS can produce RTL specifications that satisfy different design constraints (refer to Section 2.2.3.1) from the same high-level specification.
- Documenting the design process, which means keeping tracks of design decisions and their effects.
- Availability of IC technology to more people, as more design expertise is moved into the synthesis task, allowing non-expert people to produce a chip.

Although *C*-based HLS has been gaining momentum to deal with the increasing design complexity and to bridge the productivity gap [Sullivan et al., 2004], transforming a *C*-based language into a hardware language has proven to be a great challenge. Edwards [Edwards, 2006] listed the key challenges of synthesizing hardware from *C*-based languages. There exist languages issues (concurrency model and types) as well as synthesis issues (specifying timing, communication patterns, hints and constraints):

1. Concurrency is fundamental for efficient hardware (each step or computation is performed by separate hardware), however *C*-based languages impose sequential semantics (each step or computation is performed by reusing a central processor).
2. Timing constraints are also required for efficient hardware, but *C*-based languages do not provide information about the execution time of each sequence of instructions.
3. Data types are another major difference between hardware and software languages. Whereas software typically operates on integer and float variables or even complex data structures, hardware requires flat structures on bit level.
4. Communication also presents a challenge. *C*'s memory model is a large undifferentiated array of bytes, yet many small varied memories are most effective in hardware. There is no dynamic memory allocation in hardware as communication channels and patterns need

to be explicit. However, Communication channels and patterns do not exist in *C*-based languages because they use pointers to dynamically allocate storage. These differences in memory structure directly impact communication mechanisms between parallel processes. Whereas software generally uses shared memory to communicate between processes, parallel processes are truly concurrent and can therefore communicate directly in hardware systems, or through dedicated hardware such as First-In First-Out (FIFO) buffers.

5. Constraints and implementation hints are the two main ways to implement a construct such as addition in hardware, but standard *C* has no such facility.

These languages and synthesis issues were addressed by introducing *C*-variants that are more adapted to hardware implementations such as SpecC [Gajski et al., 2012], HardwareC [Ku and Micheli, 1990], SystemC [Grotker, 2002] and Handel-C [Agi, 2007], as well as pragmas/directives in most of the C-to-gates tools mentioned below.
Meeus et al. [Meeus et al., 2012] carried out a comparison between twelve different commercial and open-source HLS tools based upon a set of criteria, encompassing design entry (source language, documentation, code size), tool capabilities (support for data types, optimization and verification), design implementation capabilities (ease of implementation and abstraction level) and quality of results. In the following, I cite several *C*-based HLS tools that use the *C*-to-gates paradigm and then I carefully examine a state-of-the-art HLS tool.

1. Academic *C*-based HLS tools include:

   - SPARK from University of California, San Diego, [Gupta et al., 2003] which transforms specifications in a small subset of *C* into RTL VHDL hardware models. However, there are limitations on the subset of the *C* language that SPARK accepts as input such as lack of design hierarchy (e.g. subprograms) and of "while" type of loops [Anagnostopoulos et al., 2012].
   - GAUT from the Université de Bretange Sud [Coussy et al., 2008], is a HLS tool that is designed for Digital Signal Processing (DSP) applications. However, GAUT is incapable of handling non-static loops.
   - ROCCC from University of California, Riverside, [Villarreal et al., 2010] which is a *C* to VHDL compilation tool. However, there are limitations on the subset of the *C* language that ROCCC accepts as input such as lack of generic pointers, shifting by a variable amount, non-for loops, and the ternary operator. Moreover, ROCCC does not support the CHStone benchmarks test suite [Hara et al., 2009] which is a set of 12 *C* programs for testing the performance of different *C*-based HLS tools.
   - LegUp from University of Toronto [Canis et al., 2011], is a HLS tool that accepts the full standard of *C* as input and supports the CHStone benchmarks. However, there are unsupported constructs such as dynamic memory allocation, floating point operations and recursive function calls.

   Academic HLS tools are interesting from a research point of view, however it's risky to use them in a commercial environment since they are generally unsupported in the longer term.

2. Commercial *C*-based HLS tools include Synphony *C*[5] compiler from Synopsis, Catapult C [6] from Calypto, Cynthesizer[7] from Cadence, CyberWorkbench[8] from NEC, *C*-to-Silicon[9] from Cadence and Vivado HLS[10] from Xilinx. Research in [Meeus et al., 2012] reveals an under-performance as compared to autopilot an earlier form of Vivado HLS. For these reasons, the Vivado HLS tool will be retained as the state-of-the-art HLS tool for the low-level hardware synthesis in Chapters 4 and 5 and will be detailed in the following.

[5] http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx
[6] http://calypto.com/en/products/catapult/overview
[7] http://www.cadence.com/products/sd/cynthesizer/pages/default.aspx
[8] http://www.nec.com/en/global/prod/cwb/
[9] http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx
[10] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

Vivado HLS a state-of-the-art HLS tool In 2011 Xilinx acquired the AutoPilot HLS tool developed by AutoESL, as a part of the Vivado Design Suite. Vivado HLS compiles *C*-based input languages to RTL, which can then be synthesized and implemented onto the programmable logic of a Xilinx FPGA. Only minimal *C*-code modifications are necessary to generate an RTL design. To enable concurrency, Vivado HLS provides automatic pipelining for functions and loops. Vivado HLS converts each datatype to arbitrary-precision datatypes. From there, algorithm and interface synthesis are performed on the design. Vivado HLS supports the generation of several interface types. The Vivado HLS tool also allows a fast DSE through pragmas/directives techniques in order to optimize the design according to several design constraints. After generation of the code, a design report is generated that estimate the clock period, the FPGA resource utilization, latency, and throughput of the RTL implementation. The generation procedure can be approached either using the Graphical User Interface (GUI) or command line interface with the help of Tool Command Language (Tcl) commands. Vivado HLS provides a complete *C* validation and verification environment including *C* and RTL simulation. Vivado HLS is significantly easy to learn and use. It offers three perspectives to the developer. A debug perspective where the *C*-based code can be checked for correctness. A synthesis perspective that allows the developer to run simulations, synthesize and implement designs and view reports. An analysis perspective that allows developers to analyze synthesis results after synthesis completes. It turns out that Vivado HLS has a significant advantage since it speed up productivity for the Xilinx 7 series devices (Artix-7, Kintex-7, and Virtex-7) and many generations of FPGAs to come. In [Dubey et al., 2015], authors brought out the capability of Xilinx Vivado HLS tool and elaborated a comparison with Legup based upon synthesis results of the CHStone benchmarks. Moreover, the Berkeley Design Technology Inc. (BDTI)[11] certified and published report on the Vivado HLS tool efficiency relative to an FPGA implementation created using hand-written RTL code. They concluded that the quality of the design generated with Vivado HLS is comparable with a handcrafted RTL design. These tool capabilities motivated us to consider Vivado HLS for low-level hardware synthesis in Chapters 4 and 5.

Another important issue concerns the ability of HLS design to handle complex systems such as real-time video processing algorithms. As real-time video processing systems have to operate on huge amounts of data in little time, exploitation of parallelism is crucial in order to meet real-time requirements (Section 2.2.2). Parallelism can be defined as the decomposition of the computation into smaller pieces that can be executed in parallel. It requires that the computation is parallelizable, which means either the data used for the computation, or the task of the computation can be somehow divided. We distinguish between data parallelism and task parallelism in video processing applications [Culler et al., 1999]:

- **Data parallelism**: similar operation sequences or functions are performed on elements of a large data structure.
- **Task parallelism**: entirely different calculations are performed concurrently on either the same or different data.

Unfortunately, designing real-time video processing applications with monolithic sequential C-based specifications is unsuitable for parallelism exploitation.
Another important problem arises in HLS design which is the HLS *system integration* (see paragraph 2.3.4 in Section 2.3) into a SOC-based system. HLS is a *component-based approach* that allows for high-level modeling and synthesis of hardware components, but challenges arise when integrating them to the rest of the system. Other system components, like interconnects, interfaces, CPUs, and memory controllers are typically only available in HDLs and written at RTL. Thus, to integrate an HLS design into a SOC, designers must manually connect a standard RTL interface to the RTL of the HLS design, at the RTL. This manual effort of system integration can downgrade the HLS-to-SOC flow and diminish the HLS productivity gains.
In order to help to reason about the problem of HLS system integration, the author of [Teich, 2000] introduced a model extending the Y-chart, shown in Figure 2.13. This model tries to concatenate existing abstraction levels and synthesis tasks. That is, vertical arrows indicate synthesis task in each level of abstraction and horizontal arrows indicate the transfer of information

[11] http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf

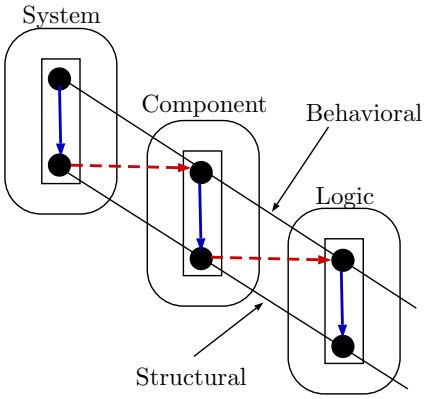


Figure 2.13: Relation between abstraction and synthesis levels [Teich, 2000].

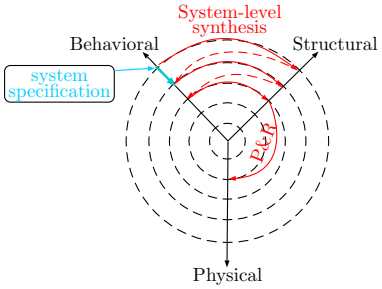


Figure 2.14: Gasjki-Kuhn Y-Chart for System-level Design Flow.

to the following lower level of abstraction. It tries also to put into perspective the system-level as a new and important abstraction level for the design of ICs, since this level holds in individual components and their interactions as well. In summary, solving the system integration problem becomes a critical design bottleneck and requires once again moving to a higher-level of abstraction. These aforementioned limitations of HLS design, combined with the limitations of HLS tools themselves, are the main reasons behind the so-called system-level design.

2.6 System-Level Design

As explained in the previous section, raising the level of abstraction to the system-level is required in order to consider complete systems instead of individual components and to explicitly express parallelism.

What. The system-level is the most abstract level. The system-level design flow starts with the system-level synthesis (Figure 2.14). System-level synthesis is the transition from a system-level specification to the algorithmic level. The result of this first synthesis step is a partitioning of the system into subsystems, a set of *communicating concurrent processes* via channels (Figure 2.15). A behavioral description at the algorithmic level for each of these subsystems (i.e. HLS)

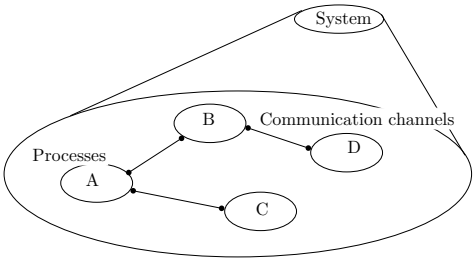


Figure 2.15: System-Level Synthesis.

ensues. RTL synthesis and P&R complete the design flow.
How. System-level design [Sangiovanni-Vincentelli, 2007] advocates the use of Model of Computations (MoCs) and the principle of separation of concerns.
What is a MoC? A MoC describes the way concurrent processes interact with each other, in an abstract way. Edward Lee, who is largely responsible for drawing attention to MoCs, describe a MoC as "the laws of physics that govern component interactions" [Lee, 2002]. Typically, MoCs are represented in a formal manner, using, for example, mathematical functions over domains, set-theoretical notations, or combinations thereof. MoCs are inherently tied to abstracted definitions of functionality, i.e., processing of data, and order, i.e., notions of time and concurrency [Gajski et al., 2009]. Edwards et al. [Edwards et al., 1997] and more recently Jantsch and Sander [Jantsch and Sander, 2005] have reviewed the MoCs used for embedded system design. In [Lee and Sangiovanni-vincentelli, 1998], Lee and Sangiovanni-Vincentelli present a framework for comparing models of computation. We distinguish between:

- Process-based models including Kahn Process Networks (KPNs) [Kahn, 1974], Dataflow Process Networks (DPNs) [Lee and Parks, 1995], Synchronous Dataflows (SDFs) [Lee and Messerschmitt, 1987] and Communicating Sequential Processs (CSPs) [Hoare, 1978];

- State-based models including Finite-State Machines (FSMs) [Gajski, 1997] and Petri Nets [Murata, 1989].

A *KPN* is a network of processes that communicate by means of unbounded FIFO queues with blocking read and nonblocking write semantics. KPNs are a popular paradigm for the description and implementation of systems for the parallel processing of streaming data. For instance, the COMPANN/LAURA approach [Stefanov et al., 2004] is based on the KPN MoC.
A *DPN* is a special case of KPNs, in which the behavior of each process (often called an actor) can be divided into a sequence of execution steps called firings by Lee and Parks [Lee and Parks, 1995]. This model is widely used in both commercial and academic projects and tools such as Synflow [Wipliez et al., 2012], the Ptolemy project from the University of California at Berkeley [Brooks et al., 2005] and Yapi [Kock et al., 2000].
A *SDF* is a special variant of DPNs where the number of values read and written by each firing of each process is constant, and does not depend on the data. The Gabriel System [Lee et al., 1989] was one of the earliest examples of a design environment that supported the SDF MoC for both simulation and code generation for DSP.
A *CSP* is an untimed MoC, in which processes synchronize based on the availability of data. Unlike KPNs, there is no storage element at all between the connected processes.
Contrary to the sequential Von Neumann MoC [Von Neumann, 1945], where both program instructions and data are kept in electronic memory, embedded system design as a MoC differs in its handling of concurrency through separation of concerns where computation and communication are separated within a system. Keutzer et al. [Keutzer et al., 2000] point out that the 'orthogonalisation of concerns' aims at breaking a complex problem into smaller, simpler pieces. In other words, a MoC is a mathematical formalism that describes the computation and communication semantics of processes independently. The computation semantics define how actors act, and the communication semantics define how they react.
MoCs are also related to languages [Edwards, 2003]. Unlike general-purpose design languages such as *C++*, SystemC, VHDL, and Verilog, system-level design languages are domain-specific. The key characteristics of Domain Specific Languages (DSLs) is their focused expressive power and being easier to write, analyze, and compile. A variety of DSLs [Hudak, 1996; van Deursen et al., 2000] have evolved, each best suited to a particular problem domain. Examples of DSLs that can be used for hardware design include Bluespec SystemVerilog (BSV)[12]and dataflow programming languages [Dennis, 1974]:

- BSV is based on a new model of computation for hardware, where all behavior is described as a set of rewrite rules, or guarded atomic actions.

- Dataflow programming is a programming paradigm that models a program as a directed graph, representing the flow of data between nodes. Some common textual dataflow programming languages include Lustre [Halbwachs et al., 1991], Signal [Benveniste et al.,

[12] http://wiki.bluespec.com/Home/BSV-Documentation

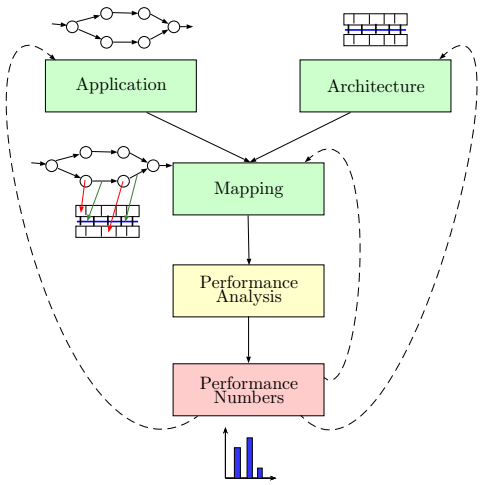


Figure 2.16: A methodology to DSE at the system-level [Kienhuis et al., 1997].

1991], StreamIT [Thies et al., 2002] and Caltrop Actor Language (CAL) [Eker and Janneck, 2003]. For example, Lustre and Signal rely on a SDF MoC. CAL relies on a DPN MoC.

Why. The advantages of the system-level design includes: a) correctly and easily program complex systems b) Raise design team productivity c) maximize design reuse d) explore design space and appraise design metrics at early stage.
At the system-level, design reuse is improved and the opportunities for DSE are expanded by developing the application separately from the architecture (as demonstrated in the Y-chart), and then selecting a mapping of the application onto the architecture, as depicted in Figure 2.16. The performance of the mapped system is then evaluated. If it is found satisfactory, then the design is finished. Otherwise, three different elements can be modified, the application, the architecture, and the mapping of the application onto the architecture.

Dataflow programming for video processing As explained throughout this chapter, most video processing algorithms are inherently parallel because they involve similar computations for all pixels in an image. The highly computational and data-parallel nature of video processing algorithms is well managed by hardware parallelism. Moreover, I have found that modeling parallelism, both task and data parallelism, is mandatory for increasing the performance of video processing algorithms. Since sequential video processing code is notoriously difficult to parallelize, and since concurrency is explicitly exposed using a dataflow graph at a higher level of abstraction, dataflow languages map well onto video processing algorithms [Sen et al., 2008; Bhartacharyya et al., 2000] which can be directly implemented in FPGAs.
The next chapter is devoted to more detailed analysis of the dataflow approach, more precisely the CAL dataflow approach, to representing embedded video compression algorithms from the system-level.

2.7 Conclusion

Over the years, video compression algorithms have improved in their compression efficiency at the expense of increased computational complexity. The design of such complex applications poses challenges, due to the high volume of video data that need to be processed and especially when hard real-time requirements are needed. For example, at real-time video rates of 25 frames per second a single operation performed on every pixel of a 768×576 PAL frame equates to 33 million operations per second. Nowadays, embedded systems are proven to successfully address the computational complexity and real-time challenges, due to the inherent parallelism capabilities they offer. The overall goal of IC design is two-fold: satisfying functional requirements while optimizing design constraints. That is why, coding/decoding of the video, is now an important issue in the field of embedded systems. However, the exponential increase in the number of

transistors on a chip, known as Moore's Law, has lead to rapidly increasing the complexity of IC design. This leads us to the well-known productivity gap –forecast by the ITRS– which is the result of the disparity between the rapid paces at which design complexity has increased in comparison to that of design productivity.
In order to shrink this productivity gap, EDA and CAD tools are required. Throughout this chapter, raising the levels of abstraction was proved to be a viable way to better cope with the design productivity gap. I have found that HDL-based and *C*-based design flows have increased designer's productivity. However, on one hand, the traditional HDL-based design flow is time consuming and lack flexibility in terms of ease of modification. On the other hand, the *C*-based design flow do not provide systematic facilities for modeling concurrency from a sequential specification. Additionally, many HLS tools only generate individual hardware components that the user still needs to integrate into a system design manually. The above mentioned limitations require even higher level of abstraction by introducing the parallelism on the system-level, instead of in the algorithmic-level. I have concluded the capability of MoCs and DSLs for explicitly exploiting parallelism and the efficiency of dataflow languages to implement video decoders. Therefore, in 2008, the MPEG used a dataflow language (CAL) to describe their new video standard, the Reconfigurable Video Coding (RVC), which will be the subject of the next chapter.

Bibliography

[Agi, 2007] *Handel-C Language Reference Manual*. Agility, 2007. 20, 27

[Amos et al., 2011] D. Amos, A. Lesea, and R. Richter. *FPGA-based Prototyping Methodology Manual*. Synopsys Press, 2011. 8, 27

[Anagnostopoulos et al., 2012] I. Anagnostopoulos, M. Bieliková, P. Mylonas, and N. Tsapatsoulis. *Semantic Hyper/Multimedia Adaptation: Schemes and Applications*. Springer Berlin Heidelberg, 2012. 20, 27

[Andriamisaina et al., 2010] C. Andriamisaina, P. Coussy, E. Casseau, and C. Chavet. High-level synthesis for designing multimode architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(11):1736–1749, 2010. ix, 18, 19, 27

[Arcas-Abella et al., 2014] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014. 17, 27

[Bailey, 2011] D. Bailey. *Design for Embedded Image Processing on FPGAs*. Wiley, 2011. 9, 27

[Bell and Newell, 1971] C. Bell and A. Newell. *Computer structures: readings and examples*. McGraw-Hill, 1971. 11, 27

[Benveniste et al., 1991] A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics . *Science of Computer Programming*, 16(2):103 – 149, 1991. 23, 27

[Bhartacharyya et al., 2000] S. Bhartacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for signal processing systems. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 47(9):849–875, Sep 2000. 24, 27

[Bossen et al., 2012] F. Bossen, B. Bross, K. Suhring, and D. Flynn. HEVC Complexity and Implementation Analysis. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22 (12):1685–1696, Dec 2012. 10, 27

[Brooks et al., 2005] C. Brooks, E. A. Lee, X. Liu, Y. Zhao, H. Zheng, S. S. Bhattacharyya, C. Brooks, E. Cheong, M. Goel, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Y. Zhao, and H. Zheng. Ptolemy II - heterogeneous concurrent modeling and design in Java, 2005. 23, 27

[Bross et al., 2012] B. Bross, W.-J. Han, G. Sullivan, and T. Ohm, J.-R. andWiegand. High Efficiency Video Coding (HEVC) Text Specification Draft 9, 2012. 9, 27

[Canis et al., 2011] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM. 20, 27

[Coussy and Morawiec, 2008] P. Coussy and A. Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008. 18, 27

[Coussy and Takach, 2009] P. Coussy and A. Takach. Guest editors' introduction: Raising the abstraction level of hardware design. *IEEE Design and Test of Computers*, 26:4–6, 2009. 19, 28

[Coussy et al., 2008] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin. GAUT: A High-Level Synthesis Tool for DSP Applications. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 147–169. Springer Netherlands, 2008. 20, 28

[Culler et al., 1999] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/-software Approach*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Morgan Kaufmann Publishers, 1999. 21, 28

[Dennis, 1974] J. B. Dennis. First Version of a Data Flow Procedure Language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 362–376, London, UK, UK, 1974. Springer-Verlag. 23, 28

[Dougherty and Laplante, 1995] E. R. Dougherty and P. A. Laplante. *Introduction to Real-Time Imaging*. Wiley-IEEE Press, 1st edition, 1995. 7, 28

[Dubey et al., 2015] A. Dubey, A. Mishra, and S. Bhutada. Comparative Study of CH Stone Benchmarks on Xilinx Vivado High Level Synthesis Tool. *International Journal of Engineering Research & Technology (IJERT)*, 4:237–242, January 2015. 21, 28

[Edwards, 2003] S. Edwards. Design Languages for Embedded Systems. Technical report, Columbia University, May 2003. 23, 28

[Edwards, 2006] S. Edwards. The challenges of synthesizing hardware from c-like languages. *Design Test of Computers, IEEE*, 23(5):375–386, May 2006. 19, 28

[Edwards et al., 1997] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3): 366–390, Mar 1997. 23, 28

[Eker and Janneck, 2003] J. Eker and J. W. Janneck. CAL Language Report Specification of the CAL Actor Language. Technical report, EECS Department, University of California, Berkeley, 2003. 24, 28

[Gajski, 1997] D. Gajski. *Principles of Digital Design*. Prentice Hall, 1997. 23, 28

[Gajski and Kuhn, 1983] D. Gajski and R. Kuhn. Guest Editors' Introduction: New VLSI Tools. *Computer*, 16(12):11–14, Dec 1983. 7, 11, 28

[Gajski et al., 2010] D. Gajski, T. Austin, and S. Svoboda. What input-language is the best choice for high level synthesis (HLS)? In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 857–858, June 2010. 18, 28

[Gajski et al., 2012] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SPECC: Specification Language and Methodology*. Springer US, 2012. 20, 28

[Gajski et al., 2009] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated, 1st edition, 2009. 13, 23, 28

[Gries and Keutzer, 2006] M. Gries and K. Keutzer. *Building ASIPs: The Mescal Methodology*. Springer US, 2006. 16, 28

[Grotker, 2002] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. 20, 28

[Gupta et al., 2003] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466, Jan 2003. 20, 28

[Halbwachs et al., 1991] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991. 23, 29

[Hara et al., 2009] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *JIP*, 17:242–254, 2009. 20, 29

[Henzinger and Sifakis, 2006] T. A. Henzinger and J. Sifakis. The Embedded Systems Design Challenge. In *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*, pages 1–15. Springer, 2006. 10, 29

[Hoare, 1978] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, Aug. 1978. 23, 29

[Hudak, 1996] P. Hudak. Building Domain-specific Embedded Languages. *ACM Comput. Surv.*, 28 (4es), Dec. 1996. 23, 29

[IEEE 1076-200] IEEE 1076-200. IEEE Standard VHDL Language Reference Manual, Jan 2009. 16, 29

[IEEE 1463-2001] IEEE 1463-2001. IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language, Oct 1996. 16, 29

[ISO/IEC 11172-2:1993] ISO/IEC 11172-2:1993. Information technology – Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s – Part 2: Video. 9, 29

[ISO/IEC 14496-2:1999] ISO/IEC 14496-2:1999. Information technology – Coding of audio-visual objects – Part 2: Visual. 9, 29

[ITU-T Rec. , 12/90] ITU-T Rec. (12/90). H.261 : Video codec for audiovisual services at p x 64 kbit/s. 9, 29

[ITU-T Rec. H.263 , 03/96] ITU-T Rec. H.263 (03/96). H.263 : Video Coding for Low bit rate Communication. 9, 29

[ITU-T Rec. H.264 and ISO/IEC 14496-10] ITU-T Rec. H.264 and ISO/IEC 14496-10. H.264 : Advanced Video Coding for Generic Audiovisual Services, May 2003. 9, 29

[Jacobs and Probell, 2007] M. Jacobs and J. Probell. A Brief History of Video Coding. *ARC International Whitepaper*, 2007. ix, 9, 29

[Jansen, 2003] D. Jansen. *The Electronic Design Automation Handbook*. Springer, 2003. 15, 29

[Jantsch and Sander, 2005] A. Jantsch and I. Sander. Models of Computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, March 2005. Special issue on Embedded Microelectronic Systems; Invited paper. 23, 29

[Kahn, 1974] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam. 23, 29

[Keutzer et al., 2000] K. Keutzer, a.R. Newton, J. Rabaey, and a. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000. 23, 29

[Kienhuis et al., 1997] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pages 338–349, July 1997. ix, 13, 24, 29

[Kock et al., 2000] E. A. D. Kock, G. Essink, W. J. M. Smits, and P. V. D. Wolf. YAPI: Application modeling for signal processing systems. In *In Proc. 37th Design Automation Conference (DAC2000*, pages 402–405. ACM Press, 2000. 23, 30

[Ku and Micheli, 1990] D. Ku and G. D. Micheli. HardwareC - A Language for Hardware Design Version 2.0. Technical report, Computer Systems Lab, Stanford Univ., 1990. 20, 30

[Lavagno et al., 2006] L. Lavagno, G. Martin, and L. Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC Press, Inc., Boca Raton, FL, USA, 2006. 11, 30

[Lee and Parks, 1995] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83 (5):773–801, May 1995. 23, 30

[Lee, 2002] E. A. Lee. Embedded Software. In *Advances in Computers*, volume 56. Academic Press, 2002. 23, 30

[Lee and Messerschmitt, 1987] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245, Sept. 1987. 23, 30

[Lee and Sangiovanni-vincentelli, 1998] E. A. Lee and A. Sangiovanni-vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, 1998. 23, 30

[Lee et al., 1989] E. A. Lee, E. Goei, H. Heine, W.-H. Ho, S. Bhattacharyya, J. C. Bier, and E. Guntvedt. GABRIEL: A Design Environment for Programmable DSPs. In *DAC*, pages 141–146, 1989. 23, 30

[Martin et al., 2010] G. Martin, B. Bailey, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Systems on Silicon. Elsevier Science, 2010. 18, 30

[Martinez et al., 2008] D. Martinez, R. Bond, and M. Vai. *High Performance Embedded Computing Handbook: A Systems Perspective*. CRC Press, 2008. 17, 30

[Marwedel, 2006] P. Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 7, 30

[McFarland et al., 1990a] M. C. McFarland, A. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, Feb 1990a. 18, 30

[McFarland et al., 1990b] M. C. McFarland, A. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78:301–318, Feb 1990b. 19, 30

[Meeus et al., 2012] W. Meeus, K. VanBeeck, T. Goedem, J. Meel, and D. Stroobandt. An overview of todays high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012. 18, 20, 30

[Michel et al., 2012] P. Michel, U. Lauther, and P. Duzy. *The Synthesis Approach to Digital System Design*. Springer US, 2012. 13, 30

[Mohanty et al., 2008] S. Mohanty, N. Ranganathan, E. Kougianos, and P. Patra. *Low-Power High-Level Synthesis for Nanoscale CMOS Circuits*. Springer US, 2008. 12, 30

[Moore, 1965] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. 7, 30

[Murata, 1989] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989. 23, 30

[Neuendorffer and Vissers, 2008] S. Neuendorffer and K. Vissers. Streaming Systems in FPGAs. In M. Berekovic, N. Dimopoulos, and S. Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5114 of *Lecture Notes in Computer Science*, pages 147–156. Springer Berlin Heidelberg, 2008. 7, 30

[Null and Lobur, 2010] L. Null and J. Lobur. *Essentials of Computer Organization and Architecture*. Jones and Bartlett Publishers, Inc., USA, 3rd edition, 2010. 12, 18, 31

[Palnitkar, 2003] S. Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Number vol. 1 in Verilog HDL: A Guide to Digital Design and Synthesis. SunSoft Press, 2003. 17, 31

[Sangiovanni-Vincentelli, 2003] A. Sangiovanni-Vincentelli. The tides of EDA. *Design Test of Computers, IEEE*, 20(6):59–75, Nov 2003. 15, 31

[Sangiovanni-Vincentelli, 2007] A. Sangiovanni-Vincentelli. Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. *Proceedings of the IEEE*, 95(3):467–506, March 2007. 23, 31

[Sen et al., 2008] M. Sen, Y. Hemaraj, W. Plishker, R. Shekhar, and S. S. Bhattacharyya. Model-based mapping of reconfigurable image registration on FPGA platforms. *J. Real-Time Image Processing*, 3(3):149–162, 2008. 24, 31

[Stefanov et al., 2004] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette. System design using Khan process networks: the Compaan/Laura approach. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 340–345 Vol.1, Feb 2004. 23, 31

[Stroud et al., 2009] C. E. Stroud, L.-T. L.-T. Wang, and Y.-W. Chang. {CHAPTER} 1 - introduction. In L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, editors, *Electronic Design Automation*, pages 1 – 38. Morgan Kaufmann, Boston, 2009. 16, 31

[Sullivan et al., 2004] C. Sullivan, A. Wilson, and S. Chappell. Using C based logic synthesis to bridge the productivity gap. In *Design Automation Conference, 2004. Proceedings of the ASP-DAC 2004. Asia and South Pacific*, pages 349–354, Jan 2004. 19, 31

[Teich, 2000] J. Teich. Embedded System Synthesis and Optimization. *Proc. Workshop SDA*, pages 9–22, 2000. ix, 21, 22, 31

[Thies et al., 2002] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In R. N. Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, UK, 2002. Springer-Verlag. 24, 31

[van Deursen et al., 2000] A. van Deursen, P. Klint, and J. Visser. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000. 23, 31

[Verhelst and Dehaene, 2009] M. Verhelst and W. Dehaene. *Energy Scalable Radio Design: for Pulsed UWB Communication and Ranging*. Analog Circuits and Signal Processing. Springer Netherlands, 2009. 13, 31

[Villarreal et al., 2010] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134, May 2010. 20, 31

[Von Neumann, 1945] J. Von Neumann. First draft of a report on the EDVAC. Technical report, Los Alamos National Laboratory, 1945. 23, 31

[Walker and Thomas, 1985] R. Walker and D. Thomas. A Model of Design Representation and Synthesis. In *Design Automation, 1985. 22nd Conference on*, pages 453–459, June 1985. 11, 14, 31

[Wipliez et al., 2012] M. Wipliez, N. Siret, N. Carta, and F. Palumbo. Design IP Faster: Introducing the C∼ High-Level Language. In *IP-SOC 2012 Conference*, Grenoble, FRANCE, 12/2012 2012. 23, 31

[Wolf, 2008] W. Wolf. *Computers As Components, Second Edition: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2008. ix, 14, 31

3

Dataflow Programming in the RVC Framework

Choose a job you love, and you will never have to work a day in your life.

Confucius

3.1 Introduction

Having argued in the previous chapter the efficiency of dataflow languages to implement video decoders which can be directly implemented in FPGAs, this chapter reviews at first dataflow programming in Section 3.2. The dataflow programs I consider in this thesis are dynamic dataflow programs that behave according to the DPN. The vertices in a DPN are called actors and are written with a DSL called RVC-CAL. RVC-CAL is a language that was standardized by the RVC standard, and with which video coding tools are defined. Section 3.3 introduces the RVC framework. Section 3.4 introduces the syntax and the semantics description of the RVC-CAL language used to formalize RVC specifications. Section 3.5 describes the supporting tools and discusses the drawbacks of related work from hardware implementation viewpoint.

3.2 Dataflow Programming

Dataflow programming is a programming paradigm whose execution model can be represented by a directed graph (Figure 3.1), where the nodes represent computational units, called *actors*, and the arcs represent streams of data, called *tokens*. Each node is an executable block that has data inputs, performs transformations over it and then forwards it to the next block. A dataflow application is then a composition of processing blocks, with one or more initial source blocks and one or more ending blocks, linked by a directed edge [Sousa, 2012]. Dataflow programming has been subject of study in the area of Software Engineering for more than 40 years, with its origins being traced back at the Ph.D. thesis of Sutherland [Sutherland, 1966]. Sutherland

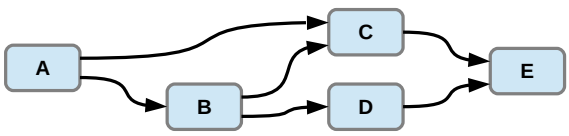
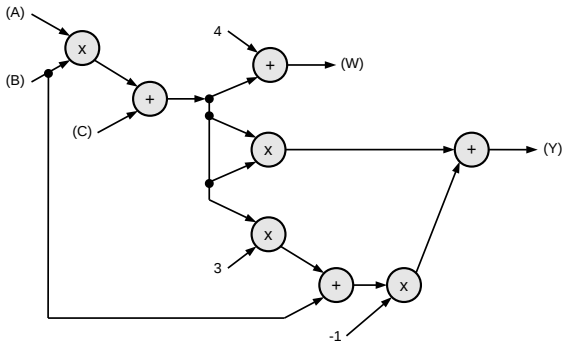


Figure 3.1: A dataflow graph containing five components interconnected using communication channels.

represents an arithmetic computation in both written and graphical forms to demonstrate the importance of the graphical form. In the written form (Figure 3.2(a)) the operations must be performed sequentially whereas in the graphical form (Figure 3.2(b)) there are three operations that could be performed simultaneously. Afterward, the first dataflow programming language and the first definition of how dataflow implementation should operate were presented by Dennis in 1974 [Dennis, 1974].

```
Z = A x B + C
W = Z + 4
Y = Z²-(3Z + B)
```

(a) Written statement.



(b) Graphical statement.

Figure 3.2: The first dataflow representation as introduced by Sutherland in 1966 [Sutherland, 1966].

In addition to the fact that the representation of video decoding applications in a set of computational units interconnected by communication channel is quite straightforward, the following features consolidate the dataflow programming to be an attractive candidate for designing parallel processing for video decoding applications.

- **Concurrency**: each node of a dataflow graph can be considered and executed independently, thus more than one operation can be executed at once. Hence it is inherently parallel and has the potential for massive parallelism. The ability of dataflow programming paradigm to express explicit concurrency makes its an alternative to the imperative sequential paradigm.

- **Scalable parallelism** [Carlsson et al., 2011]: as explained is Section 2.5, the performance of video processing applications needs to be scalable through parallelism. In parallel computing, a distinction is made between parallelism that scales with the size of the problem (data parallelism) and parallelism that scales with the size of the program (task parallelism). Dataflow has an inherent ability for parallelization. That is, scaling an algorithm over larger amounts of data is a relatively well-understood problem that applies to dataflow programs. Moreover a dataflow program has a straightforward parallel composition mechanism, unlike von Neumann programs, that tends to lead to more parallelism as applications grow in size.

- **Modularity**: the separation of concerns in system-level design (see Section 2.6) promotes modularity and allows the application to be specified in hierarchical, reusable and reconfigurable manners. Hierarchy, reusability and reconfigurability are simplified by dataflow modeling.

  - Hierarchy: a component of the network may represent another sub-network such as the component B in Figure 3.3.

  - Reusability: a single component can be used to specify several applications, or can be used several time in the network which specifies the application, such as the components A and C in Figure 3.3 that are both reused by the sub-network.

  - Reconfigurability: a component can easily be replaced by another one while its interfaces (input and output ports) are strictly identical, such as the components D and G in Figure 3.3.

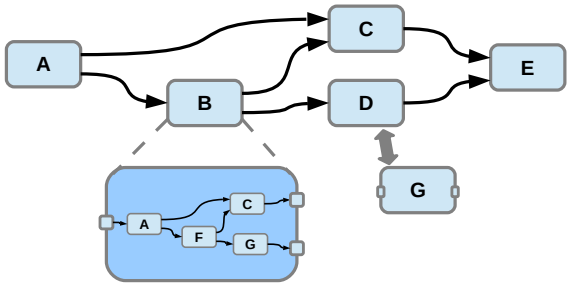


Figure 3.3: Modularity in dataflow graph.

- **Portability**: portability of video coding applications on different platforms becomes a crucial issue and such property is not appropriately supported by the traditional sequential specification model and associated methodologies. However, high-level dataflow-based descriptions aim to be compiled in lower-level languages (hardware as well as software) from an unique high-level description of the application.

For these reasons together with the limitations of the *C*-based monolithic specifications discussed in the previous chapter, the MPEG adopted a subset of the CAL dataflow programming language as a specification language for video coding and decoding algorithms within the RVC framework.

3.3 The RVC Standard

3.3.1 Motivation and Objectives

Based on the dataflow paradigm, the ISO/IEC MPEG committee standardized the RVC [Mattavelli et al., 2010] framework in 2009 for the specification of video codecs. The goal of the RVC effort is to address the limitations of monolithic specifications (usually in the form of *C*/*C*++ programs) that no longer cope with the increased complexity of video coding standards and hide the inherent parallelism of such data-driven, i.e., streaming applications. Moreover, such monolithic specifications do not enable designers to exploit the commonalities between the different video codecs (Section 2.2.2) and to produce timely specifications. The main objective of the RVC standard is to make video codecs more reconfigurable, meaning that different codecs with different configurations (e.g., different video coding standards, different profiles and/or levels, different system requirements) can be build on the basis of a unified library of video coding algorithms instead of monolithic algorithms [Lucarz et al., 2009].

3.3.2 Structure of the Standard

Two standards are defined within the context of the MPEG RVC framework:

- ISO/IEC23001-4 [ISO/IEC 23001-4], also called MPEG-B Part 4, defines the overall framework as well as the standard languages that are used to specify a new codec configuration of an RVC decoder including:
  - The specification of the Functional Unit (FU) Network Language (FNL), which is the language describing the network of one video decoder configuration. The FNL is an eXtensible Markup Language (XML) dialect that provides the instantiation of the FUs composing the codec, their parameterization, as well as the specification of the connections. FNL is another name for the XML Dataflow Format (XDF). For the network example of Figure 3.4, we have 2 FUs `FU_A` and `FU_B`. The corresponding XDF code is presented in Listing 3.1. Each vertex or edge from the graphical representation (Figure 3.4) corresponds to an element of the XML-based representation (Listing 3.1). For example, the vertex `FU_A` represents one `instance` of an entity which is identified by its `Class` composed of the package name, i.e. the localization of the entity (`test.example1.`), and the name of the entity (`Algo_FU_A`). For this instantiation, the parameter `counter` is set to 0. An edge represents a `Connection`,

i.e. a communication channel, between two entities. An FNL defines 3 types of edges: (1) between an input port of a network and an instance (*input*) (2) between an output port of an instance and an input port of another instance (*Connection*) (3) between an output port of an instance and the output port of a network (*Output*). A connection may also be parametrized with a specific channel size. FNL allows also hierarchical constructions, i.e. a FU can be described as a composition of other FUs.

- The specification of the RVC-Bitstream Syntax Description Language (BSDL) [ISO/IEC 23001-5], which is a subset of the standard MPEG BSDL, a language syntactically describing the structure of the input encoded bitstream.
- The specification of the RVC-CAL, the language that is used to express the behavior of each FU (Section 3.4).

- ISO/IEC23002-4 [ISO/IEC CD 23002-4], also called MPEG-C Part 4, specifies a normative standard library of video coding algorithms employed in the current MPEG standards, the Video Tool Library (VTL). VTL represents each coding tools from MPEG standards as one FU. Each FU has a textual specification that provides its purpose and a reference implementation expressed in RVC-CAL.

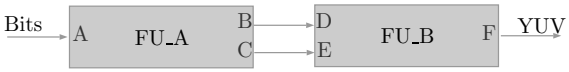


Figure 3.4: RVC network example.

```
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="Example">
<Input src="FU_A" src-port="A"/>
 <Instance id="FU_A">
  <Class name="test.example1.Algo_FU_A"/>
   <Parameter name="counter">
      <Expr kind="Literal" literal-kind="Integer" value="0"/>
   </Parameter>
 </Instance>
 <Instance id="FU_B">
  <Class name="test.example1.Algo_FU_B"/>
 </Instance>
 <Connection src="FU_A" src-port="B" dst="FU_B" ds -port="D"/>
<Connection src="FU_A" src-port="C" dst="FU_B" dst-port="E"/>
<Output src="FU_B" src-por ="F"/>
</XDF>
```

Listing 3.1: XDF code of Figure 3.4

3.3.3 Instantiation Process of a RVC Abstract Decoder Model (ADM)

Figure 3.5 depicts the process of instantiating an ADM in the RVC framework. In the normative part, the concept of the RVC framework revolves around the idea of associating a *decoder description* –combining a Bitstream Syntax Description (BSD) written in the RVC-BSDL with the FU Network Description (FND) written in the FNL– to the encoded video bitstream. The *decoder configuration* process takes place by constructing the syntax parser (built from the BSD), and the network of FUs (built from the FND) that is carried out by interconnecting coding tools from the VTL. Proprietary FUs i.e. not standardized in MPEG-C part 4 can be added to a decoder configuration as long as they respect the MPEG-B part 4 paradigm. The outcome of this configuration process is a normative behavioral CAL model of a Profile of a decoder in a MPEG standard, namely the ADM. This configuration corresponds to an oriented graph where vertices are the required FUs and edges are the communication dependencies between FUs. Figure 3.4 gives an example of a decoder configuration. The ADM is the conformance point of a RVC decoder specification. Once the ADM is specified, it is up to the users to derive the implementations of the ADM using nonnormative tools (Section 3.5) for direct and efficient synthesis targeting hardware or multi-core platforms [Gorin et al., 2013].

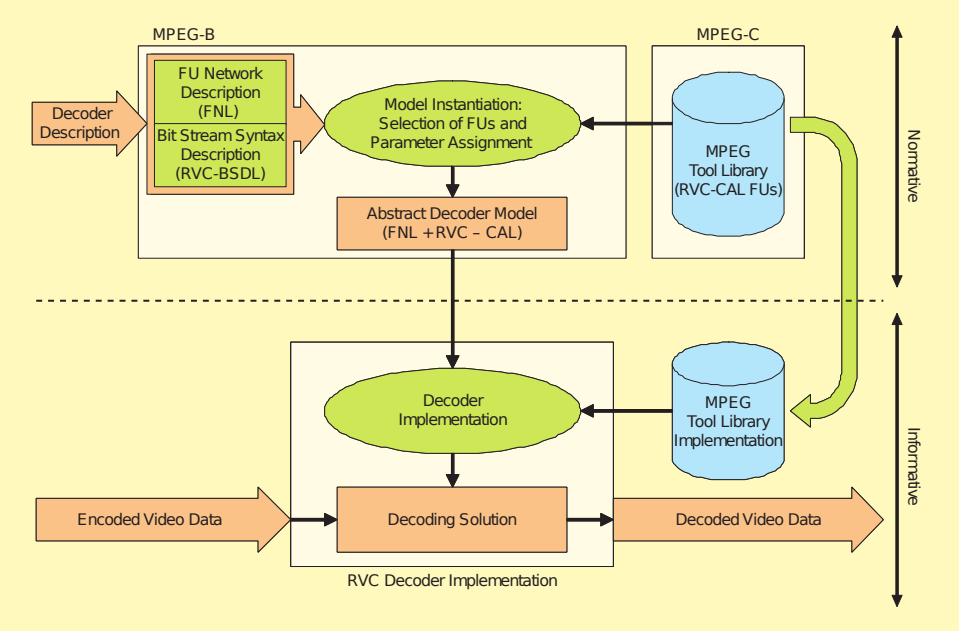


Figure 3.5: The conceptual process of deriving a decoding solution by means of normative and nonnormative tools in the RVC framework [Mattavelli et al., 2010].

3.4 RVC-CAL Dataflow Programming

This section presents the RVC-CAL language and covers the syntax (Subsection 3.4.1) and semantics, i.e. the different MoCs that can be represented with the language (Subsection 3.4.2).

3.4.1 RVC-CAL Language

CAL [Eker and Janneck, 2003; Eker et al., 2003] is a dataflow- and actor-oriented language that was developed and initially specified as a subproject of the Ptolemy project at the University of California at Berkeley. RVC-CAL, a subset of the original CAL language, is a DSL that has been standardized by MPEG RVC as the reference programming language for describing FUs' behavior. An actor in RVC-CAL represents an instantiation of an RVC FU.

Actor Structure An RVC-CAL actor is an entity that is conceptually separated into an header and a body.

- The header describes the name, parameters, and port signature of the actor. For instance, the header of the actor shown in Listing 3.2 defines an actor called `Adder`. This actor takes one boolean parameter whose value is specified at runtime, when the actor is instantiated, i.e. when it is initialized by the network that references it. The port signature of `Adder` is two input ports `A` and `B` and an output port `C`.

```
1 actor Adder(bool checkValue) int A, int B ==> int C:
2
3   //body
4
5 end
```

Listing 3.2: Header of an RVC-CAL Actor.

- The body of the actor may be empty, or may contain *state variables* declarations, *functions*, *procedures*, *actions*, *priorities*, and at most one *FSM*:

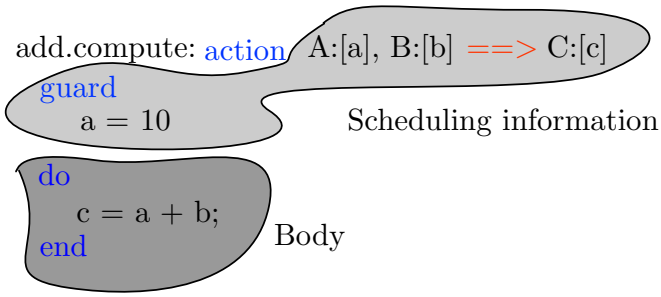


Figure 3.6: Scheduling information and body of an action.

– State variables can be used to define constants and to store the state of the actor they are contained in. The first four lines of Listing 3.3 shows the three different ways of declaring a state variable.

– RVC-CAL supports the common concepts that are traditionally used by procedural languages such as functions and procedures. Listing 3.3 shows an example of a function and a procedure declaration.

```
1  //State variables
2  int coeff = 32;//initiliaze a constant
3  uint (size =4) num_bits := 0;//initialize a variable
4  uint(size =16) bits ;
5
6  //Functions
7  function abs(int(size=32) x) --> int(size=32) :
8    if(x > 0) then x else -x end
9  end
10
11 //Procedures
12 procedure max (int a, int b)
13 var
14   int result := 0
15 begin
16    if(a > b) then
17     result := a;
18    else
19     result := b;
20    end
21 end
```

Listing 3.3: State variables, functions and procedures declarations in RVC-CAL.

– The only entry points of an actor are its actions; functions and procedures can only be called by an action. An action corresponds to a *firing function*, which describes, in a procedural manner, the behavior of the actor during action's execution or *firing*. Figure 3.6 shows the syntax of an action definition. An action may be identified by a *tag*, which is a list of identifiers separated by colons, where t_a denotes the tag of action a (e.g., `add.compute`). The scheduling information, that defines the criteria for action to fire, involves:

  1. the patterns of tokens read and written by a single action which are called *input pattern* and *output pattern*;
  2. firing conditions, called *guards* or *peek pattern*, that constraint action firings according to the values of incoming tokens and/or state variables. Note that guard conditions can "peek" at the incoming tokens without actually consuming them.

  The contents of an action, that are not scheduling information, are called its body, and define what the action does. The body of an action is like a procedure in most imperative programming languages, with local variables and imperative statements. Statements may be conditionals (`if/then/else`), loops (`for/while`), calls to functions and procedures, and assignments to local and state variables.

– Priorities establish a partial-order between action tags. They have the form $t_1 > t_2 > ... > t_n$. Priorities define the order in which actions are tested for schedulability (lines 9 – 11 of Listing 3.4).

– An FSM regulates the action firings according to state transitions in order to describe the internal scheduling of an actor (lines 12 – 17 of Listing 3.4).

```
1  actor Abs() int(size=16) I
2  ==> uint(size=15) O, uint(size=1) S:
3    pos : action I:[u] ==> O:[u] end
4    neg : action I:[u] ==> O:[  u ]
5      guard u < 0
6    end
7    unsign : action ==> S:[0] end
8    sign : action ==> S:[1] end
9    priority
10     neg > pos;
11   end
12   schedule fsm s0:
13     s0 (pos) --> s1;
14     s1 (unsign) --> s0;
15     s0 (neg) --> s2;
16     s2 (sign) --> s0 ;
17   end
18 end
```

Listing 3.4: An RVC-CAL actor example with priority and FSM.

Type System The type system is one of the major differences between the original CAL and RVC-CAL. Whereas CAL keeps an abstract type system authorizing untyped data, RVC-CAL defines a practical type system dedicated to the development of signal processing algorithms, including:

- A logical data type which has two potential values `true` and `false` and is declared using the keyword `bool`.
- Bit-accurate integer data types: an integer can be signed or unsigned, declared with the `int` and `uint` keywords respectively. Moreover, the bit-width may be omitted, in which case the type has a default bit-width, or it can be specified by an arbitrary expression. For instance the type `int(size=8)` considers a signed integer coded on 8 bits.
- Floating-point types coded with 16, 32 and 64 bits, that are declared respectively using the `half`, `float` and `double` keywords.
- A type to describe a sequence of characters, `String`.
- A list type that behaves more like an array type, and is declared with a given type and size, such as `List(type:int,size=8)` that represents a list of 8 integers.

3.4.2 Representation of Different MoCs in RVC-CAL

RVC-CAL Semantics Figure 3.7 illustrates the principles of the RVC-CAL dataflow programming model. In this model, FUs are implemented as actors containing a number of actions and internal states. An actor is a modular component that encapsulates its own state. The state of any actor is not shareable with other actors. Thus, an actor cannot modify the state of another actor. The absence of shared state allows the actors to execute their actions while avoiding *race conditions* on their state. Interactions between actors are only allowed through FIFO channels, connected between ports of actors. The behavior of an actor is defined in terms of a set of actions. The actions in an actor are atomic, which means that once action fires no other action can fire until the previous is finished. An action firing is an indivisible quantum of computation that corresponds to a mapping function of input tokens to output tokens. This mapping is composed of three ordered and indivisible steps:

- consume input tokens;

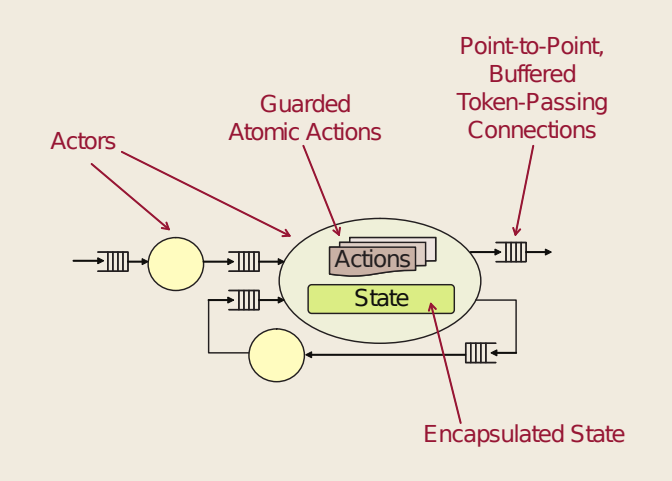


Figure 3.7: Pictorial representation of the RVC-CAL dataflow programming model [Amer et al., 2009].

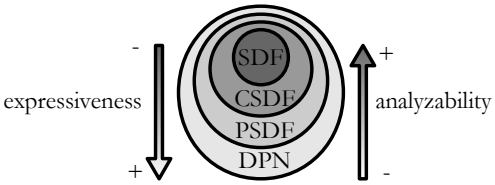


Figure 3.8: Classification of dataflow MoCs with respect to expressiveness and analyzability [Wipliez and Raulet, 2010].

- modify internal state;
- produce output tokens.

Each action of an actor may fire depending on four different conditions:

1. input token availability (i.e. there are enough tokens on all its input ports);
2. guard conditions (that evaluate actor's internal state and peek into the input tokens' value);
3. FSM based action scheduling;
4. action priorities (that describe which action shall be fired for when multiple actions are eligible to fire).

The topology of a set of interconnected actors constitutes what is called a network of actors. At the network level, the actors can work concurrently, each one executing their own sequential operations. RVC-CAL also allows hierarchical system design, in which each actor can be specified as a network of actors.
A MoC , i.e. the interpretation of a network of actors, determines its semantics-it determines the result of the execution, as well as how this result is computed, by regulating the flow of data as well as the flow of control among the actors in the network [Eker and Janneck, 2003].

Denotational Semantics of the MoC of RVC-CAL The dataflow model standardized in MPEG RVC is based on the DPN [Lee and Parks, 1995]. This model is selected since it is the most expressive model among other dataflow models, such as Parameterized Synchronous Dataflow (PSDF) [Bhattacharya and Bhattacharyya, 2001], Cyclo-Static Dataflow (CSDF) [Bilsen et al., 1995] or SDF [Lee and Messerschmitt, 1987] (Figure 3.8). Of course, the RVC-CAL supports implementations of actors that can have a behavior that is data- and state-independent i.e. static (SDF), state-dependent i.e. cyclo-static (CSDF), data-dependent i.e. quasi-static (PSDF), or data- and state-dependent i.e. dynamic (DPN). In this thesis, my main focus is on the dynamic

behavior.
RVC-CAL dataflow model respects the semantics of DPN. I define here the denotational semantic used in [Lee and Parks, 1995]. DPNs are shown to be a special case of KPNs [Kahn, 1974]. A DPN program is a network of dataflow actors that communicate through unidirectional FIFO channels with unbounded capacity. DPNs are described by a graph $G = (V, E)$, where V is a set of dataflow actors, and E is a set of FIFO channels. Each channel $e \in E$ carries a possibly infinite sequence of tokens denoted $X = [x_1, x_2, ...]$, where each x_i is a token. We denote the empty sequence as $\perp$. We write $X \subseteq Y$ to say sequence X is a prefix of sequence Y. E.g. $[x_1, x_2] \subseteq [x_1, x_2, x_3]$. The set of all sequences is denoted as S and the set of n-tuples of sequences on the n FIFO channels of an actor is denoted as S^n, that is $\mathbf{X} = \{X_1, X_2, ..., X_n\} \in S^n$. Examples of elements of S^2 are $s1 = [[x_1, x_2, x_3], \perp]$ or $s2 = [[x_1], [x_2]]$. The length of a sequence is given by $|X|$, similarly the length of an element $s \in S^n$ is in turn noted as $|s| = [|X_1|, |X_2|, ..., |X_n|]$. For instance, $|s1| = [3, 0]$ and $|s2| = [1, 1]$. For actor $\alpha \in V$, the sets $inports(\alpha) = \{1, 2, ..., p\}$ and $outports(\alpha) = \{1, 2, ..., q\}$ denote input and output ports. Lee [Lee and Parks, 1995] extends the KPN principle by introducing the notion of firing. Execution of a DPN is a possibly infinite execution of its actors. The execution of a DPN actor is a sequence of atomic firings. Firings of the actor can be represented with a firing function that maps a set of input sequences to a set of output sequences, such as $f : S^p \rightarrow S^q$. Actor α may have multiple firing functions $F_\alpha = \{f_1, f_2, .., f_n\}$ where $f_i : S^p \rightarrow S^q$ for $1 \leq i \leq n$. Each firing function $f_i \in F_\alpha$ has an associated firing rule. Hence, actor α has n firing rules $R_\alpha = \{R_1, R_2, ..., R_n\}$, one for each firing function. A firing function is enabled if and only if its firing rule is satisfied. A firing rule $R_i \in R_\alpha$ is a finite sequence of patterns and specifies rules for each of the p input ports, given as $R_i = [P_{i1}, P_{i2}, ..., P_{ip}] \in S^p$. Each pattern P_{ij} is an acceptable sequence of tokens in R_i on one input j from the input p of an actor. For firing rule R_i to be satisfied, each pattern P_{ij} must form a prefix of the sequence of unconsumed tokens at input port j as $P_{ij} \subseteq X_j$, for all $j = 1, ...p$. The pattern $P_{ij} = \perp$ is satisfied for any sequence, whereas the pattern $P_{ij} = [*]$ is satisfied for any sequence containing at least one token. The symbol $*$ denote a token wildcard. E.g. consider an adder with two inputs. It has only one firing rule, $R_1 = \{[*], [*]\}$, meaning that each of the two inputs must have at least one token. Conclusively, instead of the context switching found in the KPN model, the DPN can be executed by continuously scheduling actor firings at run-time according to firing rules (dynamic scheduling).
I have stated that one essential benefit of the DPN model lies in its strong expressive power, so as to simplify algorithm implementation for programmers and create efficient implementations. This expressive power includes:

- *The ability to describe data- and state-dependent behaviors*: We call a firing rule *data-dependent* if it has a rule whose patterns depend on values of input tokens. We call an actor data-dependent if it has a data-dependent firing rule. We call a firing rule *state-dependent* if it has a rule whose patterns depend on the value of the state of the actor. We call an actor state-dependent if it has a state-dependent firing rule. State and data dependencies allow us to implement *dynamic* actors whose input and output rates vary between firings. To do so, the RVC-CAL language extends the DPN MoC by adding a notion of guard to firing rules. Formally the guards of a firing rule are boolean predicates that may depend on the input patterns, the actor state, or both, and must be `true` for a firing rule to be satisfied. We define the guards of a firing rule with predicates that return a set of valid sequences. Predicates are associated to the patterns of the rule so that G_{ij} is the guard predicate associated to the jth pattern of R_i. An interesting example of a data-dependent actor is the `Select` actor in Listing 3.5, which has the firing rules $\{R_1, R_2\}$, where

$$R_1 = \{[*], \perp, [T]\} \tag{3.1}$$

$$R_2 = \{\perp, [*], [F]\} \tag{3.2}$$

  where T and F match *true* and $false$-valued booleans.

```
1 actor Select() int(size=8) A, int(size=8) B, bool S ==> int(size=8) output↩
    :
2   action A:[v], S:[sel] ==> output:[v]
```

```
3     guard sel
4   end
5   action B:[v], S:[sel] ==> output:[v]
6     guard not sel
7   end
8 end
```

Listing 3.5: The `Select` actor in RVC-CAL.

- *The ability to produce time-dependent behaviors*: DPN places no restrictions on the description of actors, and as such it is possible to describe a time-dependent actor in that its behavior depends on the time when the tokens are available. An actor is time-dependent when a lower priority action requires fewer tokens than a higher priority action and their guard expressions are not mutually exclusive [Wipliez and Raulet, 2012].

- *The ability to express non-determinism*: DPN adds non-determinism to the KPN model, by allowing actors to test an input port for the absence or presence of data. Indeed, in a KPN process, writes to a FIFO are non-blocking (they always succeed immediately), but reads from a FIFO are blocking. This means that a process that attempts to read from an empty input channel stalls until the buffer has sufficient tokens to satisfy the read. Conversely, in a DPN actor, reads from a FIFO are non-blocking. This means an actor will only read data from a FIFO if enough data is available, and a read returns immediately. As a consequence, an actor need not be suspended when it cannot read. Listing 3.6 shows an example of a non-determinate merge in RVC-CAL. Its behavior consists in moving tokens whenever they arrive on any of its two inputs to its unique output. Hence, the output sequence depends on the arrival times of the input tokens.

```
1 actor merge() int(size=8) A, int(size=8) B ==> int(size=8) output:
2   action A:[v] ==> output:[v] end
3   action B:[v] ==> output:[v] end
4 end
```

Listing 3.6: The `merge` actor in RVC-CAL.

3.5 RVC-CAL Code Generators and Related Work

The DPN MoC defined in the previous section makes it possible to get efficient implementations of RVC specifications whatever the platform targeted. As explained in Section 3.3, the RVC framework is informatively supported by several tools, which are code generators. These code generators take the RVC ADM as input and generates respectively $C/C++$ code for software targets and HDL code for hardware targets [Eker and Janneck, 2012].
CAL is historically supported by a Java interpreter integrated in Ptolemy II [Buck et al., 2002] and in Moses [Esser and Janneck, 2001]. However, these two environments are currently updated by the Open Dataflow environment (OpenDF) and the Open RVC-CAL Compiler (Orcc). I reveal subsequently how an RVC-CAL dataflow program can be compiled to various target languages, and emphasize on related work from the hardware implementation point of view and their drawbacks. Figure 3.9 summarizes the existing hardware code generation tools supporting RVC.

3.5.1 OpenDF

The OpenDF tool chain is open source and released under the Berkeley Software Distribution (BSD) license and consists of a simulator and compilers for hardware and software [Bhattacharyya et al., 2008]. It is composed by two main parts: the front-end parses the CAL program and generates an Intermediate Representation (IR) called XML Language-Independent Model (XLIM). XLIM is an XML based format that represents CAL actors, at a level which is close to machine instructions, in Static Single Assignment (SSA) form. The back-end then translates XLIM into either C or HDL code:

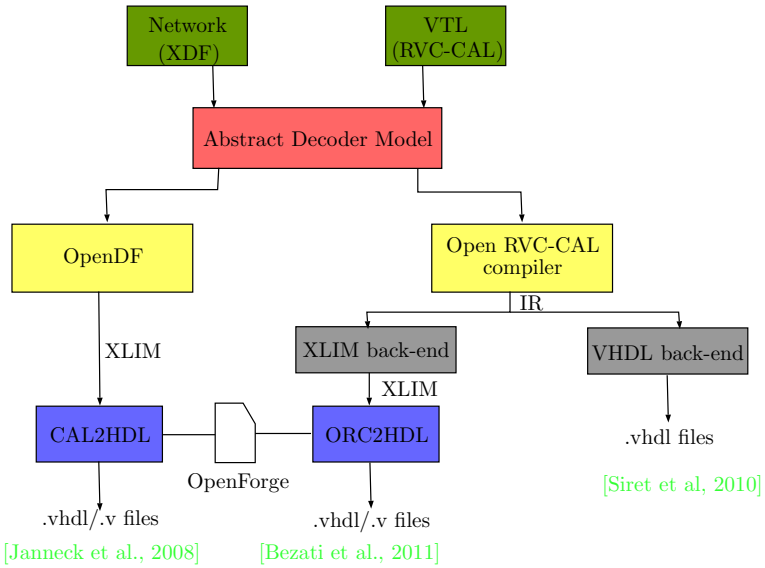


Figure 3.9: Non-standard tools for the automatic hardware code generation in the RVC framework.

- OpenDF integrates a back-end, *Xlim2C* [von Platen, 2009], developed by Ericson as part of the ACTORS project[1], which translates an XLIM representation to a *C* program dedicated to embedded platforms based on ARM processor. The tool flow from CAL to *C* in OpenDF is also known as CAL2ARM as mentionned in workpackage D1b (CAL Mapping Tools) of the ACTORS project.

- *OpenForge* acts as a back-end tool to generate a HDL representation from an XLIM one, that targets Xilinx FPGAs. The tool flow from CAL to HDL in OpenDF is also known as CAL2HDL [Janneck et al., 2008]. Each actor is translated separately into HDL and is connected with FIFO buffers in the resulting RTL descriptions. That is, the final description is made up of a Verilog file for each actor and a VHDL file for the top: the highest hierarchical representation of the design connections. The main issue with the CAL2HDL tool chain is that the code generation does not support all the RVC-CAL structures (unsigned integer types, procedures, loops and multi-token actions) and the generated code is so difficult to manage and correct.
  With the goal to overcome these issues, the OpenDF framework gave way to the Orcc framework. I concentrate on the Orcc compiler in the next subsection as it is the basis of my work.

3.5.2 Orcc

Started by Wipliez in 2009 [Wipliez, 2010], Orcc is an open-source toolkit for RVC-CAL dataflow programs. Orcc is the successor of a first version of software code generator CAL2C [Roquier et al., 2008a,b; Wipliez et al., 2008]. Orcc[2] is a complete Integrated Development Environment (IDE) based on Eclipse that embeds two editors for both actor and network programming, a simulator, a debugger and a multi-target compiler. The primary purpose of Orcc is to provide developers with a compiler infrastructure to allow software/hardware code to be generated from RVC-CAL descriptions. The compilation procedure of Orcc is shown in Figure 3.10. The first stage of the compiler, called **front-end**, is responsible of transforming RVC-CAL actors to an IR of actors, which includes steps such as parsing, typing, semantic checking, and various transformations. Parsing is done using the Xtext[3] framework [Efftinge and Völter, 2006]. The **middle-end** is the component that analyzes and transforms the IR actors and networks to produce optimized IR actors and networks. The last stage of the compilation infrastructure is

[1] www.actors-project.eu
[2] http://orcc.sourceforge.net/.
[3] Xtext is available at http://www.eclipse.org/Xtext/.

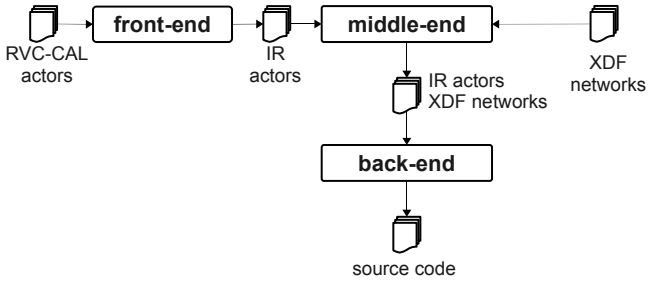


Figure 3.10: Compilation infrastructure of Orcc [Wipliez, 2010].

code generation, in which the **back-end** for a given language (e.g. *C*) generates code from a hierarchical network and a set of IR actors. The back-end for a language L is called the "L back-end" (e.g. *C* back-end). Orcc does not generate assembly or executable code directly, rather it generates source code that must be compiled by third-party tools, such as compilers and synthesizers for this language. The code generation process is different for each back-end. The different steps in the general code generation process, that a back-end can do, are listed below:

1. **Actor code generation**: The first step is the transformations undergone by the IR of actors, either generic transformations such as optimizations, or language-specific transformations necessary to generate code in a given language from the IR.

2. **Network code generation**: The second step is the transformations of the network, which consist of closing the network by replacing parameters by their concrete values, flattening a hierarchical network, and explicitly implementing broadcasts from a single output port to several input ports.

3. **Printing Code**: The last step of code generation is printing code from IR actors and networks. It transforms an IR to a target language L in a textual form using a template-based [Parr, 2004] *pretty printer* to automatic code formatting, Xtend [4], which is a simplified programming language based on Java and fully integrated within Eclipse.

Orcc has currently several built-in back-ends that use the same specific IR:

- The *C* **back-end** [Wipliez, 2010; Wipliez et al., 2011; Yviquel, 2013] produces an application described in portable ANSI *C* with multi-core ability.

- The **Low Level Virtual Machine (LLVM) back-end** [Gorin et al., 2011] generates LLVM code for actors that can then be loaded on-demand along with an XDF network by the Just-in-time Adaptive Decoder Engine (JADE).

- The **Transport-Trigger Architecture (TTA) back-end** [Yviquel et al., 2013] implements a full co-design for embedded multi-core platforms based on the TTA and generates the software code executed on the processors using the TTA-based Co-design Environment (TCE) as well as the hardware design that executes it.

- The **XLIM back-end** generates VHDL description using the OpenForge back-end. The hardware code generator presented in more details in [Bezati et al., 2011] generates a hardware description from CAL by translating Orcc's IR to XLIM, and then compiling XLIM to a HDL. The tool flow from CAL to HDL in Orcc is also known as ORC2HDL. The restriction of this methodology is the lack of the support of multi-rate RVC-CAL programs (i.e. the repeat construct is not supported). Although the solution proposed by Jerbi et al. [Jerbi et al., 2012] to overcome this limitation, which is an automated transformation of multirate RVC-CAL programs to single-rate programs, it leads to a complex resulting code and performance reduction. Recent work [Bezati et al., 2013] enhanced the ORC2HDL design flow, by directly feeding into OpenForge the IR's Orcc, known as Xronos. The main issue with this approach is the need to change some constructs in the initial RVC-CAL code to be able to synthesize it.

[4] http://www.eclipse.org/xtend/.

- The **VHDL back-end**: Another approach proposed by Siret et al. [Siret et al., 2010] offers a new VHDL code generator by adding a new back-end to Orcc. Unfortunately, the work lacked loop support and it was not finalized.

The Orcc project also maintains a repository of dataflow applications available for download[5].

3.6 Conclusion

As discussed throughout Chapter 2, the drawbacks of existing video standards specifications (whether with HDL-based or *C*-based languages) motivated the emergence of system-level design of embedded systems and revived the interest on dataflow programming for designing embedded systems. In this context, the MPEG RVC standard has emerged as a new specification formalism to design a video decoder at a system-level of abstraction by adopting the RVC-CAL dataflow programming language, that behaves according to the DPN MoC. The RVC-CAL language presents interesting features such as parallelism scalability, modularity and portability. Moreover, the DPN MoC has the advantage of explicitly exposing concurrency and modeling dynamic behavior, as found in video processing applications. While Chapter 3 described the limitations of related work on the hardware implementation of an RVC ADM, the next chapter describes the contributions of my thesis, that is to say the efficient and optimized hardware implementation of dynamic dataflow programs in the RVC framework.

[5] https://github.com/orcc/orc-apps

Bibliography

[Amer et al., 2009] I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J. F. Nezan, and O. Deorges. Reconfigurable Video Coding on multicore : an overview of its main objectives. *IEEE signal Processing Magazine, special issue on Signal Processing on Platforms with Multiple Cores, Part 1 : Overview an Methodology*, Volume 26(Issue 6):pp 113 – 123, 2009. ix, 40, 47

[Bezati et al., 2011] 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. 44, 47

[Bezati et al., 2013] E. Bezati, M. Mattavelli, and J. Janneck. High-level synthesis of dataflow programs for signal processing systems. In *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*, pages 750–754, Sept 2013. 44, 47

[Bhattacharya and Bhattacharyya, 2001] B. Bhattacharya and S. Bhattacharyya. Parameterized Dataflow Modeling for DSP Systems. *Trans. Sig. Proc.*, 49(10):2408–2421, Oct. 2001. ISSN 1053-587X. doi: 10.1109/78.950795. URL http://dx.doi.org/10.1109/78.950795. 40, 47

[Bhattacharyya et al., 2008] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet. OpenDF - A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems. In *Multi-Core Computing. MCC 2008. First Swedish Workshop on*, page CD, Ronneby, Sweden, Nov. 2008. 42, 47

[Bilsen et al., 1995] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, May 1995. doi: 10.1109/ICASSP.1995.479579. 40, 47

[Buck et al., 2002] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Readings in hardware/software co-design. chapter Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, pages 527–543. Kluwer Academic Publishers, Norwell, MA, USA, 2002. ISBN 1-55860-702-1. 42, 47

[Carlsson et al., 2011] A. Carlsson, J. Eker, T. Olsson, and C. von Platen. Scalable parallelism using dataflow programming. In *Ericson Review*. On-Line Publishing, 2011. 34, 47

[Dennis, 1974] J. B. Dennis. First Version of a Data Flow Procedure Language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 362–376, London, UK, UK, 1974. Springer-Verlag. 34, 47

[Efftinge and Völter, 2006] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Eclipsecon Summit Europe 2006*, Nov. 2006. 43, 47

[Eker and Janneck, 2012] J. Eker and J. Janneck. Dataflow programming in cal – balancing expressiveness, analyzability, and implementability. In *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*, pages 1120–1124, Nov 2012. 42, 47

[Eker and Janneck, 2003] J. Eker and J. W. Janneck. CAL Language Report Specification of the CAL Actor Language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2003/4186.html. 37, 40, 47

[Eker et al., 2003] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003. URL http://chess.eecs.berkeley.edu/pubs/488.html. 37, 48

[Esser and Janneck, 2001] R. Esser and J. Janneck. Moses-a tool suite for visual modeling of discrete-event systems. In *Human-Centric Computing Languages and Environments, 2001. Proceedings IEEE Symposia on*, pages 272–279, 2001. 42, 48

[Gorin et al., 2011] J. Gorin, M. Wipliez, F. Prêteux, and M. Raulet. Llvm-based and scalable mpeg-rvc decoder. *J. Real-Time Image Process.*, 6(1):59–70, Mar. 2011. 44, 48

[Gorin et al., 2013] J. Gorin, M. Raulet, and F. Prêteux. MPEG Reconfigurable Video Coding: From specification to a reconfigurable implementation. *Signal Processing: Image Communication*, 28(10):1224 – 1238, 2013. doi: 10.1016/j.image.2013.08.009. URL https://hal.archives-ouvertes.fr/hal-01068867. 36, 48

[ISO/IEC 23001-4] ISO/IEC 23001-4. Mpeg systems technologies part 4: Codec configuration representation, 2011. 35, 48

[ISO/IEC 23001-5] ISO/IEC 23001-5. Information technology - mpeg systems technologies - part 5: Bitstream syntax description language, 2008. 36, 48

[ISO/IEC CD 23002-4] ISO/IEC CD 23002-4. Information technology - mpeg video technologies - part 4: Video tool library, 2010. 36, 48

[Janneck et al., 2008] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs: An mpeg-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 287–292, Oct 2008. 43, 48

[Jerbi et al., 2012] K. Jerbi, M. Raulet, O. Deforges, and M. Abid. Automatic generation of synthesizable hardware implementation from high level rvc-cal description. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1597–1600, March 2012. 44, 48

[Kahn, 1974] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam. 41, 48

[Lee and Parks, 1995] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83 (5):773–801, May 1995. 40, 41, 48

[Lee and Messerschmitt, 1987] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245, Sept. 1987. 40, 48

[Lucarz et al., 2009] C. Lucarz, I. Amer, and M. Mattavelli. Reconfigurable video coding: Objectives and technologies. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 749–752, Nov 2009. doi: 10.1109/ICIP.2009.5414275. 35, 48

[Mattavelli et al., 2010] M. Mattavelli, I. Amer, and M. Raulet. The Reconfigurable Video Coding Standard [Standards in a Nutshell]. *Signal Processing Magazine, IEEE*, 27(3):159–167, May 2010. ISSN 1053-5888. doi: 10.1109/MSP.2010.936032. ix, 35, 37, 48

[Parr, 2004] T. J. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th International Conference on World Wide Web*, pages 224–233, New York, NY, USA, 2004. ACM. 44, 48

[Roquier et al., 2008a] G. Roquier, M. Wipliez, M. Raulet, J. Janneck, I. Miller, and D. Parlour. Automatic software synthesis of dataflow program: An mpeg-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 281–286, Oct 2008a. 43, 48

[Roquier et al., 2008b] G. Roquier, M. Wipliez, M. Raulet, J.-F. Nezan, and O. Deforges. Software synthesis of cal actors for the mpeg reconfigurable video coding framework. In *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, pages 1408–1411, Oct 2008b. 43, 49

[Siret et al., 2010] N. Siret, M. Wipliez, J.-F. Nezan, and A. Rhatay. Hardware code generation from dataflow programs. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 113–120, Oct 2010. 45, 49

[Sousa, 2012] T. B. Sousa. Dataflow Programming Concept, Languages and Applications. In *Doctoral Symposium on Informatics Engineering*, 2012. 33, 49

[Sutherland, 1966] W. R. Sutherland. *The On-Line Graphical Specification of Computer Procedures*. PhD thesis, Massachusetts Institute of Technology, 1966. ix, 33, 34, 49

[von Platen, 2009] C. von Platen. Cal arm compiler. Technical report, D2c, 2009. 43, 49

[Wipliez, 2010] M. Wipliez. *Compilation infrastructure for dataflow programs*. PhD thesis, INSA of Rennes, December 2010. ix, 43, 44, 49

[Wipliez and Raulet, 2010] M. Wipliez and M. Raulet. Classification and transformation of dynamic dataflow programs. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 303–310, Oct 2010. doi: 10.1109/DASIP.2010.5706280. ix, 40, 49

[Wipliez and Raulet, 2012] M. Wipliez and M. Raulet. Classification of Dataflow Actors with Satisfiability and Abstract Interpretation. *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 3(1):49–69, 2012. URL https://hal.archives-ouvertes.fr/hal-00717361. 42, 49

[Wipliez et al., 2008] M. Wipliez, G. Roquier, M. Raulet, J.-F. Nezan, and O. Deforges. Code generation for the mpeg reconfigurable video coding framework: From cal actions to c functions. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 1049–1052, June 2008. 43, 49

[Wipliez et al., 2011] M. Wipliez, G. Roquier, and J. F. Nezan. Software Code Generation for the RVC-CAL Language. *Journal of Signal Processing Systems*, 63(2):203–213, May 2011. URL https://hal.archives-ouvertes.fr/hal-00407950. 44, 49

[Yviquel, 2013] H. Yviquel. *From dataflow-based video coding tools to dedicated embedded multicore platforms*. PhD thesis, Université Rennes 1, Oct. 2013. URL https://tel.archives-ouvertes.fr/tel-00939346. 44, 49

[Yviquel et al., 2013] H. Yviquel, J. Boutellier, M. Raulet, and E. Casseau. Automated design of networks of Transport-Triggered Architecture processors using Dynamic Dataflow Programs. *Signal Processing: Image Communication*, 28(10):1295 – 1302, Sept. 2013. 44, 49

Part II

CONTRIBUTIONS

4

Toward Efficient Hardware Implementation of RVC-based Video Decoders

> *You will come to know that what appears today to be a sacrifice will prove instead to be the greatest investment that you will ever make.*
>
> Gordon B. Hinckley

4.1 Introduction

One of the research areas of the Image team of the Institute of Electronics and Telecommunications of Rennes (IETR) laboratory concerns the development of *rapid prototyping* methodologies for parallel and embedded platforms. There are two central themes in rapid prototyping [Cooling and Hughes, 1989]. The first one is a *model* to describe the behavior and the requirements of systems. The second one is automatic methods and *tools* to *quickly* generate system prototypes from the system models. Generating new prototypes and analyzing their characteristics allow developers to identify critical issues of the prototype, and then to *iteratively* improve and refine the developed embedded system.
In this chapter, we propose a fully automated design flow for rapid prototyping of RVC-based video decoders, whereby a system-level design specified in RVC-CAL dataflow language is quickly translated to a hardware implementation. Section 4.2 highlights the drawbacks of the XLIM back-end and formulates our research issue. Section 4.3 details the proposed rapid prototyping methodology. Section 4.4 presents rapid prototyping implementation results on the HEVC video decoder.

4.2 Limitations of the Current Solution and Problem Statement

As related in Section 3.5.2, although hardware code generation from RVC-CAL dataflow programs has been first presented with the OpenDF framework [Janneck et al., 2008], the work of [Bezati et al., 2011] presents an approach for unified hardware and software synthesis starting from the same RVC-CAL specification. The purpose of this work is to use Orcc and generate an XLIM code which is directly synthesizable with OpenForge. Figure 4.1 illustrates this compilation flow. That is, the Orcc's IR undergoes first a set of transformations such as inlining of RVC-CAL functions and procedures, SSA transformations, Cast Adder and so on. After

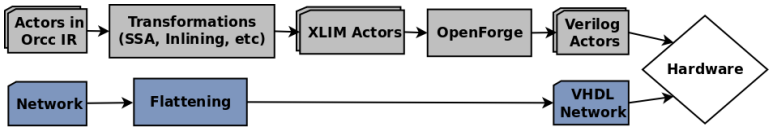


Figure 4.1: The compilation flow of the XLIM back-end [Bezati et al., 2011].

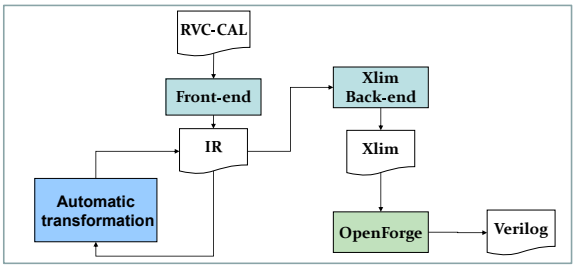


Figure 4.2: Automatic M2M tokens transformation localization in the hardware generation flow [Jerbi et al., 2012].

these transformations, the actors are printed in XML format respecting XLIM properties using a template engine called StringTemplate [Parr, 2004]. Then, OpenForge takes as input the XLIM file and generates a Verilog file that represents the RVC-CAL actor with an asynchronous handshake-style interface for each of its ports. Orcc generates a Top VHDL that connects the Verilog generated actors with back-to-back or using FIFO buffers into a complete system.
However, one main limitation of the OpenForge synthesizer is the fact that it does not support I/O multi-tokens reads/writes that consume/produce more than one token per execution. Multi-tokens reads and writes are supported by the "repeat" construct in RVC-CAL, as highlighted in Listing 4.1 where the actor "sum" consumes 5 tokens in its input and produces only one in its output.

```
1  actor sum() int(size=8) IN ==> int(size=8) OUT :
2    add : action IN:[ i ] repeat 5  ==> OUT :[s]
3      var
4        int s : = 0
5    do
6      foreach int k in 0 .. 4 do
7        s := s + i[k] ;
8      end
9    end
10 end
```

Listing 4.1: A "sum" actor written in RVC-CAL with the "repeat" construct.

To overcome this issue, the work of [Jerbi, 2012] consists in automatically transforming the data read/write processes from multi-tokens to mono-token while preserving the same actor behavior. That is, the transformation detects the multi-tokens patterns of the actor and automatically substitutes them with 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 as well as internal buffers associated with read/write indexes. All these required actions, variables and FSMs are both directly created and optimized in the IR of Orcc before generating the XLIM as shown in Figure 4.2. This aspect of the transformation localization in the conception flow is very important since it can be applied on any back-end of Orcc, even if the resulting changes in the IR are intended for hardware generation.

Although this solution has solved the main issue of the hardware generation flow from RVC-CAL programs using Orcc and OpenForge, it may add excessive sequentialization of equivalent actors states, resulting in overall performance and resource usage efficiency reduction. Moreover, the code generation process was quite slow which will raise problems when dealing with more

complex design than the AVC/H.264 decoder. However, it is valuable to investigate how to support an efficient hardware implementation of the new emerging HEVC decoder from the system-level. That is why this dissertation attempts to provide an alternative to the XLIM back-end, which is rather a close to gate representation, and to the OpenForge synthesizer.

4.3 Rapid Prototyping Methodology

This section gives an overview of our rapid prototyping methodology to provide an efficient hardware implementation of RVC-based video decoders. The aim of this work is to address the limitations of the existing approaches to the prototyping of RVC-based video decoders as complex as the HEVC decoder.

4.3.1 Outline of the Prototyping Process

As argued in Chapter 2, raising the level of abstraction to the system-level proved to be the solution for closing the productivity gap in embedded system design. A well-defined design flow enables interoperability and design automation for synthesis and verification in order to achieve the required productivity gains. Our methodology is inspired by the Gajski Y-chart and the Kienhuis Y-chart approaches (Figures 2.14 and 2.16) for *system design* and *DSE*. It consists of a rapid prototyping implementation path from a system-level model based on the RVC-CAL programming language down to synthesized system model and eventually a system prototype [Abid et al., 2013]. The synthesized system model is rapidly generated using software and hardware compilation tools. That is, the methodology combines the RVC-CAL compiler Orcc, and the *C*-to-gate tool Vivado HLS from Xilinx. The reasons that govern our choice of the *C*-to-gate tool was explained in Section 2.5. An overview of the proposed system-level design flow is outlined in Figure 4.3. A typical system-level design flow is separated into two parts: a front-end and a back-end. The following are the main steps of the presented methodology.

(a) The system design front-end (Orcc) takes a description of the application and target architecture at its input. That is, applications are given in the form of a DPN MoC that describes the actors behavior in RVC-CAL programming language. Target architectures are given in the form of an XDF file that describes the network of one video decoder configuration. The RVC-CAL actors together with the XDF network form the ADM as explained in Paragraph 3.3.3 of Section 3.3. The front-end encompasses automatic refinement for both computation and communication separately as advocated by system-level design in Section 2.6. At the output of the front-end, code generation then translates the RVC-CAL description into the target language. We chose a subset of *C* as a target language which is compliant with Vivado HLS. For this issue, we develop a new Orcc back-end that we denoted the *C*-HLS back-end. Figure 4.4 depicts stage (a) in more details.

(b) In the back-end, *C*-based component models are synthesized down to RTL components in the form of standard VHDL code such that they can feed into traditional logic and physical synthesis processes. *C*-to-RTL HLS is provided by Vivado HLS synthesizer. Vivado HLS and its coding styles are deeply discussed subsequently. Figure 4.5 depicts the Vivado HLS design flow in more details.

(c) In the end, the desired result at the output of a system-level design flow is a physical system prototype that is ready for further manufacturing. That is, using Xilinx tools a gate-level description is created by logic synthesis from a RTL model. Then, physical synthesis automates the placement of the gates in the FPGA and the routing of the inter-connections between gates from a gate-level description, which produce a full FPGA configuration bitstream.

When implementing our proposed system design flow, a number of questions arise: how to preserve the DPN semantics when implementing DPN MoCs down to *C*-based code, how to generate a *C*-based code compliant/synthesizable with Vivado HLS, how to preserve the abstract system architecture in the system implementation, how to perform automatic validation and verification of the functionality and performance of the complete design for automatic Design-Space Exploration (DSE). We will try to answer to all these questions in the following subsections.

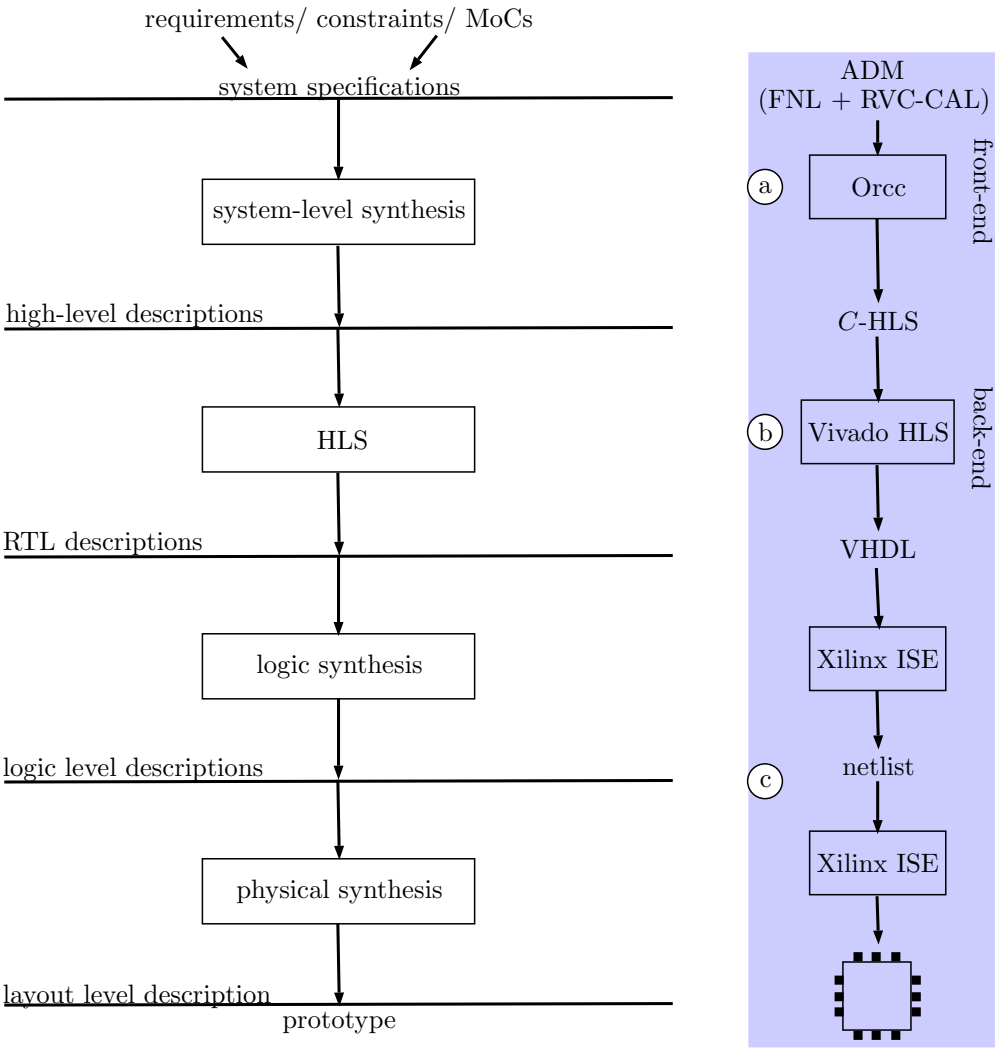


Figure 4.3: Our proposed system-level design flow.

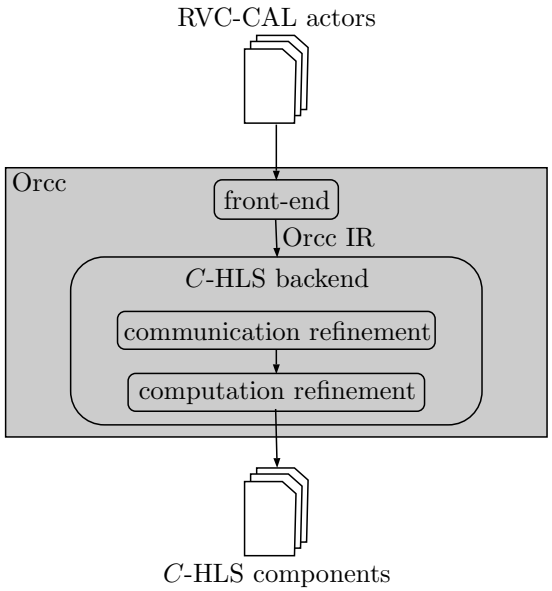


Figure 4.4: System-level synthesis stage (a) of Figure 4.3.

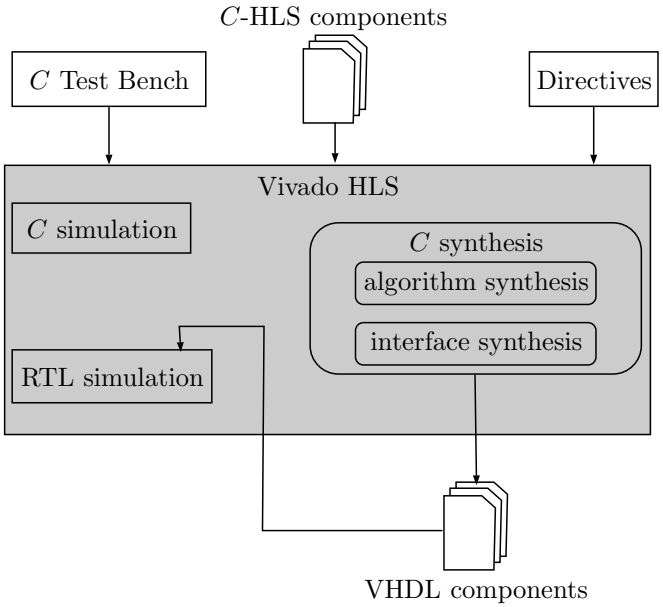


Figure 4.5: HLS stage (b) of Figure 4.3.

4.3.2 System-Level Synthesis using Orcc

This section details the system-level synthesis stage (a) of Figure 4.3. First we refer to the Vivado HLS User Guide [XIL, 2014] for details on the supported and unsupported *C* constructs by Vivado HLS, to be able to generate code for *C*-based HLS from RVC-CAL. Then we explain the code generation process within Orcc including computation and communication refinement with respect to the DPN MoC as a further consideration. We refer to Sub-section 3.5.2 for the general code generation process within Orcc.

4.3.2.1 Vivado HLS Coding Style

While Vivado HLS supports a wide range of the *C* language, some constructs are not synthesizable, including:

- Dynamic memory allocation: Vivado HLS does not support *C*++ objects that are dynamically created or destroyed with function calls such as `malloc()`, `alloc()`, `free()`, `new` and `delete`.
- System calls: all communication with the FPGA must be performed through the input and output ports. There is no underlying Operating System (OS) (such as `time()` and `printf()`) or OS operations (such as file read/write) in an FPGA. For example, `print` statements are automatically ignored by Vivado HLS and there is no requirement to remove them from the code.
- Pointers: despite the restriction on dynamic memory allocation, pointers are well-supported by the Vivado HLS expect some cases:
  - when pointers are accessed (read or written) multiple times in the same function.
  - when using arrays of pointers, each pointer must point to a scalar or a scalar array (not another pointer).
  - Vivado HLS supports pointer casting between native *C* types but does not support general pointer casting, for example casting between pointers to differing structure types.
- Recursive functions: recursive functions cannot be synthesized.

One of the benefits of using a DSL like RVC-CAL dataflow language to generate code for *C*-based HLS is the fact it does not support such aforementioned constructs, i.e. no dynamic memory

allocation, no pointers, no system calls and no recursive functions. Additionally, RVC-CAL dataflow networks are a natural abstraction of hardware architectures by providing hierarchical, inherently parallel descriptions and by explicitly specifying interfaces and bit-accuracy (Section 3.4.1). Advantageously, Vivado HLS supports interface management and provides arbitrary precision data type libraries for modeling data types with a specific width.
In the rest of this sub-section, scheduling policies and communication mechanisms are discussed.

4.3.2.2 Automatic Communication Refinement

Due to huge amounts of data, communication synthesis is a critical issue that has to be considered extensively in order to obtain efficient implementations from system-level video processing applications. Consequently, the following paragraphs aim to address the communication refinement step within Orcc. This step refines RVC-CAL communications to high-level communications in the form of *C* code according to a particular communication mechanism.

Interface Specification At the system-level, the port signature of the `Select` actor of Listing 3.5 is three input ports `A`, `B` and `S` and one output port `output`. The *C*-HLS back-end automatically translates this port signature into interface declaration in the *C* code as depicted in Listing 4.2.

```
1 #include <hls_stream.h>
2 typedef signed char i8;//8-bit user defined type
3 // Input FIFOS
4 extern hls::stream<i8>  myStream_A;// A stream declaration
5 extern hls::stream<i8>  myStream_B;
6 extern hls::stream<bool>  myStream_S;
7 // Output FIFOs
8 extern hls::stream<i8> myStream_Output;
```

Listing 4.2: The *C* declaration of the interface ports of the actor `Select` using explicit streaming.

We chose to use an explicit streaming communication mechanism with the Vivado HLS *C++* template class `hls::stream<>` for modeling streaming data objects. That is, the *C*-HLS back-end models interface ports as external variables using `hls::stream<>` that behaves like a FIFO of infinite depth in the *C* code. There is no requirement to define the size of an `hls::stream<>`. Streaming data objects are defined by specifying the type and variable name. For example, a 8-bit integer type is defined and used to create a stream variable called myStream_A in Listing 4.2. The header file `hls_stream.h` defines the `hls::stream` *C++* class used to model streaming data.

Read and Write operations With respect to the DPN MoC semantics, accesses to an `hls::stream<>` object are non-blocking reads and writes as described in Sub-section 3.4.2 and are accomplished by means of class methods:

- Non-blocking write: this method attempts to push variable `v` into the stream myStream_Output (Listing 4.3).

```
1 i8 v;
2 myStream_Output.write_nb(v);
```

Listing 4.3: Usage of non-blocking write method.

- Non-blocking read: this method reads from the head of the stream myStream_A and assigns the values to the variable `v` (Listing 4.4).

```
1 i8 v;
2 myStream_A.read_nb(v);
```

Listing 4.4: Usage of non-blocking read method.

4.3.2.3 Automatic Computation Refinement

The computation refinement step includes the addition of details about the action firing and the scheduling of actions while preserving the DPN MoC semantics.

Action Firing As highlighted in Figure 3.6, each action consists of its scheduling information (input, output and peek patterns) and its body. In the *C*-HLS back-end, the scheduling information and body are implemented in two unrelated data structures. This separation allows the schedulability of actions to be tested in parallel when generating hardware code. The code that tests the schedulability of an action is put in a procedure (e.g., `isSchedulable_select_a()`), and the body of an action is represented as another procedure (e.g., `Select_select_a()`). As described in Sub-section 3.4.2, the interactions between firing rules and FIFO channels can be summarized with the help of two functions:

- gets the number of tokens available in a FIFO.
- peeks at a fixed number of tokens from a FIFO.

However, once data is read from an `hls::stream<>`, it cannot be read again. In other words, the information about the number of tokens in the input FIFO channel and their values is not available. That is, the peek stays however limited to the first token of the FIFO channel and thus reduces the support of dynamic dataflow programs. As described in the Vivado HLS User Guide , the use of the `hls::stream` construct forces the developer to cache the data locally. To do so, we used the automatic transformation in the core of Orcc proposed in [Jerbi, 2012], that we denote M2M tokens transformation in the rest of this manuscript. As noted in Section 4.2, we take advantage of the fact that the transformation can be applied on any back-end of Orcc. The transformation creates internal circular buffers for every input port where tokens could be stored and peeked, and managed by read and write indexes as depicted in Listing 4.5. The size of these internal buffers is the nearest power-of-two to the number of tokens read from the input pattern. The indexes and the buffer are created as global variables so they can be used by other actions.

```
1 static bool S_buffer[1];
2 static i32 readIndex_S = 0;
3 static i32 writeIndex_S = 0;
4 static i8 A_buffer[1];
5 static i32 readIndex_A = 0;
6 static i32 writeIndex_A = 0;
7 static i8 B_buffer[1];
8 static i32 readIndex_B = 0;
9 static i32 writeIndex_B = 0;
```

Listing 4.5: Internal buffers creation for every input port with indexes management.

Then, the idea is to separate the input and output patterns from the action and create mono-token actions that use these patterns. That is, an action is created (Listing 4.6) just to read data from the input stream and put it in the internal circular buffer while increment the read index ($readIndex := readIndex + n$, where n is the number of tokens read from the input pattern).

```
1 static void Select_untagged_A() {
2   i8 A_Input;
3
4   myStream_A.read_nb(A_Input);
5   A_buffer[readIndex_A & 0] = A_Input;
6   readIndex_A = readIndex_A + 1;
7 }
```

Listing 4.6: Input pattern's action creation.

Idem for the write index when consuming the data from the buffers (Listing 4.7).

```
1 static void Select_select_a() {
```

```
2   i8 v;
3
4   v = A_buffer[0 + writeIndex_A & 0];
5   myStream_Output.write_nb(v);
6   writeIndex_A = writeIndex_A + 1;
7 }
```

Listing 4.7: Output pattern's action creation.

Consequently, the difference between the read and the write indexes represents the number of available tokens in each buffer and all the firing rules of the actions are related to this difference. This index difference is important in the schedulability of the created actions. Using this methodology, the firing rules of Equations (3.1) and (3.2) are implemented in Equations (4.1) to (4.4) by using indexes management as follow:

$$P_{1,1} = [readIndex\_A - writeIndex\_A >= 1] \tag{4.1}$$

$$G_{1,3} = [S\_buffer[writeIndex\_S] = true] \tag{4.2}$$

$$P_{2,2} = [readIndex\_B - writeIndex\_B >= 1] \tag{4.3}$$

$$G_{2,3} = [S\_buffer[writeIndex\_S] = false] \tag{4.4}$$

where $P_{1,1}$ and $G_{1,3}$ are the pattern and the guard of firing rule R_1, whereas $P_{2,2}$ and $G_{2,3}$ are the pattern and the guard of firing rule R_2.

Action Scheduler Whereas the bodies of the actions are represented as a set of procedures in *C*-HLS, the guards, priorities, FSM, together with tests for input tokens availability, are represented in a special action selection procedure called the action scheduler. However, the implementation of the action scheduler in *C*-HLS differs slightly from the DPN MoC by further considering availability of output buffer space. Theoretically, the fact that writes are non-blocking poses no problem since FIFOs have an unbounded capacity as described in Sub-section 3.4.2. However, in practice, memory is limited in physical systems. Therefore the scheduler has to ensure that enough space is available in the output channels to allow the firing of the action without blocking. In order to have a correct hardware implementation, it was imperative to update the action scheduler to the Vivado HLS streams with fullness and emptiness tests. Testing actions fireability of the `Select` actor is done by an action scheduler according to Listing 4.8, which is a reformulation of the conditions to fire an action as described in Listing 3.5. The `Select` scheduler is updated to check the streams (not full or not empty) before writing or reading data.

```
1  void Select_scheduler() {
2    if (!myStream_A.empty() &&
3    isSchedulable_untagged_A()) {
4      Select_untagged_A();
5    } else if (!myStream_B.empty() &&
6    isSchedulable_untagged_B()) {
7      Select_untagged_B();
8    } else if (!myStream_S.empty() &&
9    isSchedulable_untagged_S()) {
10     Select_untagged_S();
11   }
12   if (isSchedulable_select_a() &&
13   !myStream_output.full()) {
14     Select_select_a();
15   }else if (isSchedulable_select_b() &&
16   !myStream_output.full()) {
17     Select_select_b();
18   }
19 }
```

Listing 4.8: The action scheduler of the `Select` actor.

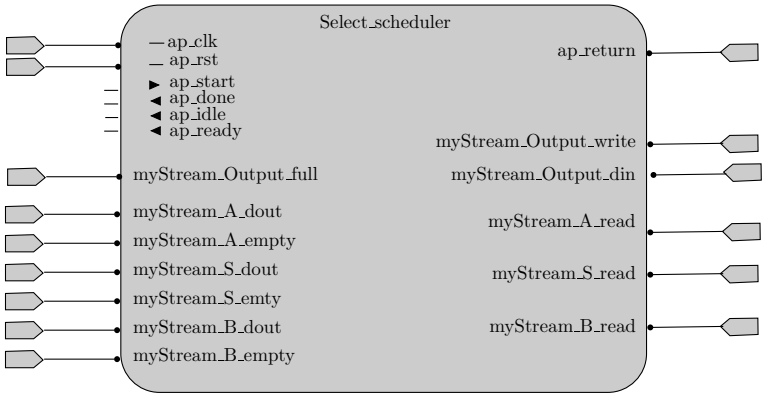


Figure 4.6: The corresponding RTL implementation of the interface ports of the actor `Select` using explicit streaming.

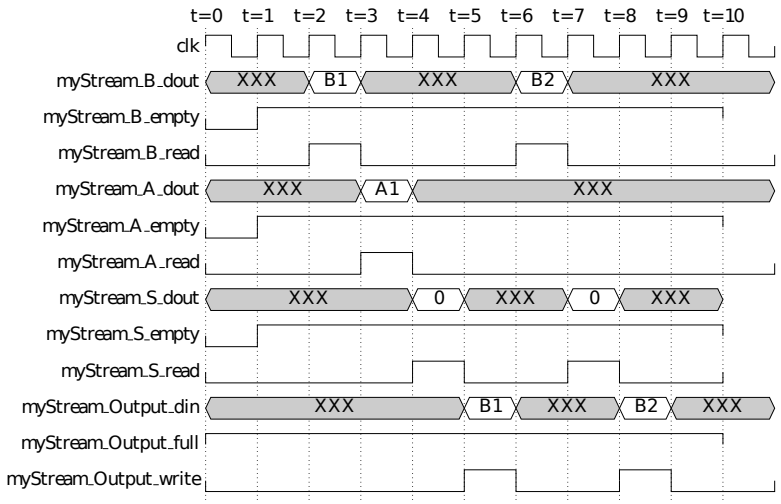


Figure 4.7: Timing behavior of `ap_fifo` interfaces port of the actor `Select`.

4.3.3 HLS using Vivado HLS

This section details the HLS stage (b) of Figure 4.3. The goal of the Xilinx Vivado HLS tool is to create RTL implementation in VHDL format for each *C*-HLS component. The Xilinx Vivado HLS includes the steps described in Figure 2.12. HLS performs two distinct types of synthesis, namely:

1. **Algorithm synthesis** takes the content of the actor and schedules the functional statements into RTL statements over a number of clock cycles. Each function is synthesized into a corresponding module or entity/architecture.

2. **Interface synthesis** operates on interface ports and transform them into RTL ports with appropriate hand-shaking signals, allowing the actor to communicate with other actors in the system. In this case, the `hls::stream<>` variables are automatically implemented as `ap_fifo` interfaces with read, write, full and empty ports. That is, when an `hls::stream` is synthesized it is automatically implemented as a FIFO channel with a depth of 1. For the `Select` actor, the *C* specification is synthesized into an RTL block with the ports shown in Figure 4.6, where the `Select_scheduler` is the top-level function for synthesis. The timing behavior is shown in Figure 4.7.

Table 4.1 summarizes additional control signals generated by default by the Vivado HLS tool for all designs. Note that Vivado HLS supports *C* simulation prior to synthesis to validate the *C* algorithm and *C*/RTL co-simulation after synthesis to verify the RTL implementation, in order to improve productivity.

Table 4.1: By default all HLS generated designs have a master control interface.

Signal	Description
ap_clk	signal from external clocking source
ap_rst	asynchronous (or synchronous) reset
ap_start	enables computation
ap_done	end of computation
ap_idle	RTL block is idle
ap_ready	RTL block is able to accept next computation
ap_return	the data in the ap_return port is valid when ap_done
din	*data inputs*
dout	*data outputs*

4.3.4 System-Level Integration

Yet, each actor is translated separately to VHDL code. In order to elaborate the system-level, we take advantage of the fact that:

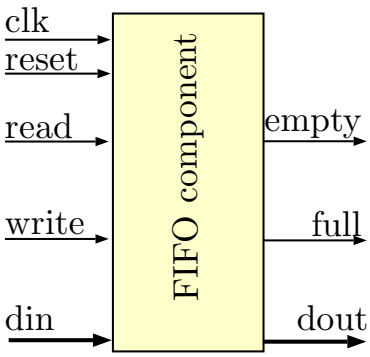
- Vivado HLS tool works well in generating hardware implementation from a unique actor at the component level as explained in the previous sub-section.
- The dataflow networks in RVC are described using an XML-based language, as depicted in Listing 3.1, that can be parsed to extract information about hand-shaking connections.

In addition, while the streams are declared and used as externals enabling the hardware component to communicate with FIFOs, the FIFOs need to be physically generated. For this issue, we modified and used a generic FIFO component defined in the literature of Vivado HLS as illustrated in Figure 4.8(a). The bit width of the FIFO is put as generic to match the bit width of the input and output data of the source and target actors. Hence, while each individual actor is synthesized to VHDL code with the appropriate hand-shaking signals, the instantiation of the VHDL actors and the connecting FIFOs is done in a top-level netlist file "Top" in VHDL generated automatically by Orcc. Thus, the system-level is obtained by connecting the VHDL components with FIFO buffers and the different actors can fire in a parallel way (*Actor scheduling*). Figure 4.8(b) depicts an example of a system-level elaboration between a source and a target actor.
For a synchronous behavior, all clocks and reset signals are connected to those of the "Top" entity.

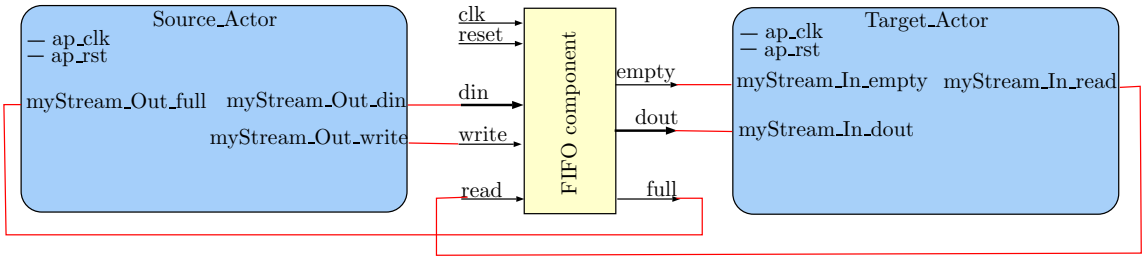
Actor Scheduling As remarked in Sub-section 3.4.2, DPNs must be scheduled dynamically, i.e. actors are scheduled at runtime by an actor scheduler. However, unlike software scheduling [Yviquel et al., 2011], we do not need to schedule the actors in hardware since all actors can run in parallel. Actors are executed concurrently, each one is managed by its own action scheduler. In other words, the actors compute their values at the same time whenever data is available on their inputs, which implies that the resulting system is fully self-scheduled pursuant to the flow of tokens.

4.3.5 Automatic Validation

Once having the system implementation, *validation* is needed to check its correctness through *simulation*. Simulation is a dynamic process to validate the functionality and the metrics of the model in terms of the execution output for given input vectors [Chen and Dömer, 2014].
For the validation of the generated design, Orcc supports automated test-bench generation for all granularity levels of the network which means that we created a test bench for each actor, each network and each sub-network. This approach revealed to be very important to accelerate debugging and assessing the hardware generated implementation at both component and system levels.



(a) The RTL implementation of FIFO buffer.



(b) An example of a system-level elaboration at the RTL between a source and a target actor using explicit streaming.

Figure 4.8: System-level elaboration using explicit streaming.

4.4 Rapid Prototyping Results: HEVC Decoder Case Study

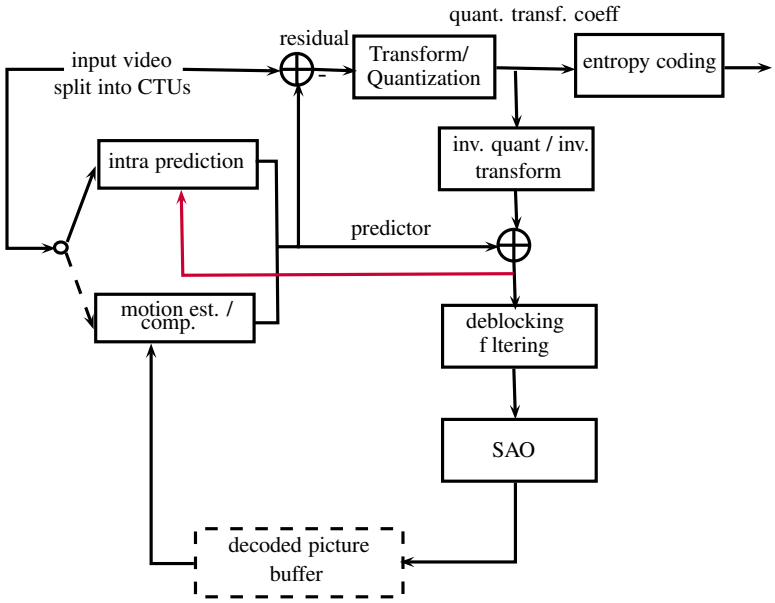
To demonstrate the applicability of our proposed rapid prototyping methodology, we automatically synthesized an RVC-CAL implementation of a HEVC decoder application into FPGA according to the system-level design flow of Figure 4.3. We chose the HEVC standard as this is the latest video coding standard of the ITU–T VCEG and the ISO/IEC MPEG in a Joint Collaborative Team on Video Coding (JCT–VC). The HEVC standard is formally known as ISO/IEC MPEG-H Part 2 (ISO/IEC 23008−2) and ITU–T H.265. Moreover, the HEVC decoder involves high computational complexity engine consuming a coded bit-stream on its input, and producing video data (samples) on its output. At 30 frames of 1080p per second, this amounts to $30 * 1920 * 1080 =$ approximately 62.2 million pixels per second. In the common YUV420 format, each pixel requires 1.5 bytes on average, which means the decoder has to produce 93.3 million bytes per second.

4.4.1 RVC-CAL Implementation of the HEVC Decoder

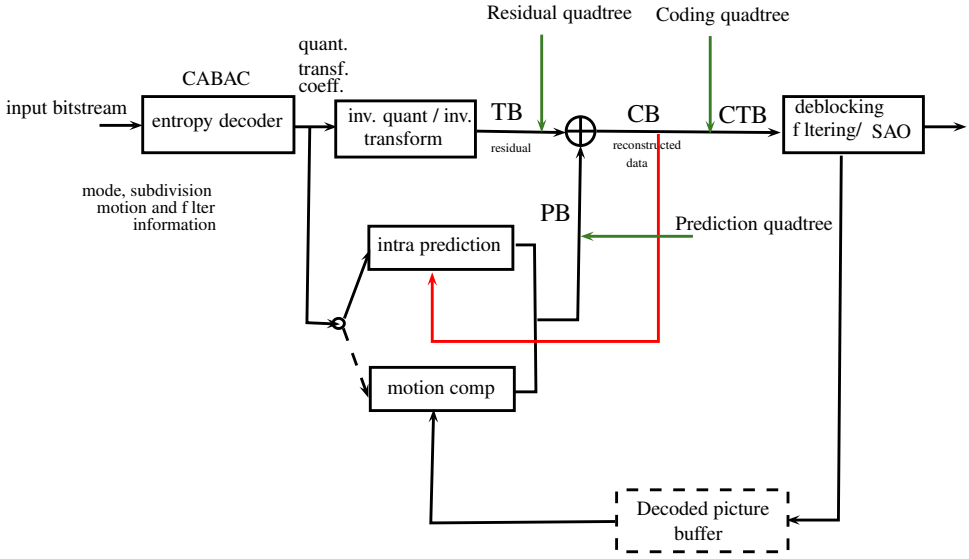
In the following, some key features of the HEVC coding design are first outlined, then the RVC-CAL implementation of the HEVC decoder used in this work is described.

4.4.1.1 The HEVC standard

The HEVC standard [Sullivan et al., 2012; Sze et al., 2014] is designed to address essentially all existing applications of H.264/MPEG-4 AVC (Sub-section 2.2.2). It aims to particularly achieve multiple goals including coding efficiency (i.e. reducing bitrate requirements by half with comparable image quality), supportability of increased video resolution and implementability using parallel processing architectures. Just like all video compression standards since H.261, the general structure of HEVC is based on the hybrid video coding scheme, illustrated in Figure 2.2, which uses transform coding for exploiting spatial redundancies and Motion Compensation (MC) for exploiting temporal redundancies. In the following, only the decoding process is in the scope of this study. Figure 4.9(b) depicts the typical block diagram of a HEVC video decoder deduced from the HEVC video encoder diagram of Figure 4.9(a). Before detailing the blocks of



(a) Typical HEVC video encoder.



(b) Block-diagram of a HEVC decoder.

Figure 4.9: HEVC encoder/decoder.

the diagram, it is noteworthy that HEVC supports the *quadtree-based block partitioning* concept [Kim et al., 2012] based on a Coding Tree Unit (CTU) instead of a macroblock. A macroblock consists of a fixed-size 16×16 block of luma samples and two corresponding 8×8 blocks of chroma samples as used in prior video standards. A CTU consists of a luma Coding Tree Block (CTB), the corresponding chroma CTBs and syntax elements. The variable-size $L \times L$ of a luma CTB can be chosen as $L = 16, 32$, or 64 samples, with a larger block size usually increasing the coding efficiency. Each CTU is partitioned into Coding Units (CUs) recursively. A CU consists of a one luma Coding Block (CU) and two chroma CUs. The decision whether to code a picture area using interpicture (temporal) or intrapicture (spatial) prediction is made at the CU level. Each CU has an associated partitioning into Prediction Units (PUs) for the purpose of prediction and into a tree of Transform Units (TUs) for the purpose of transform. Similarly, each CU is split into Prediction Blocks (PBs) and Transform Blocks (TBs). This variable-size, adaptive approach is particularly suited to larger resolutions, such as $4k \times 2k$. Related with the picture partitioning, HEVC uses different techniques to support parallel decoding and error resilience namely slices, tiles and Wavefront Parallel Processing (WPP) [Chi et al., 2012]. Slices partition a picture into groups of consecutive CTUs in raster scan order. Tiles split a picture horizontally and vertically into rectangular regions that can independently be decoded/encoded. WPP splits a picture into rows of CTUs.
A decoding algorithm receiving an HEVC compliant bit-stream on its input would typically proceed as follows.

- **Entropy Decoder**: decodes the video syntax elements within the incoming video bit-stream, using Context-Adaptive Binary Arithmetic Coding (CABAC) [Marpe et al., 2003]. CABAC in HEVC was designed for higher throughput. HEVC uses a Network Abstraction Layer (NAL) unit based bit-stream structure [Sjberg et al., 2012], which is a logical data packet where each syntax structure is placed [Sze and Marpe, 2014].

- **Inverse Quantization and Transform**: the quantized transform coefficients, that are obtained at the output of the entropy decoder, are de-quantized based on the Uniform-Reconstruction Quantization (URQ) scheme controlled by a Quantization Parameter (QP)) and inverse transformed using the Inverse Discrete Cosine Transform (IDCT), at the TU level [Budagavi et al., 2014; De Souza et al., 2014].

- **Intra-prediction**: predicts the samples of a PB according to reference samples (samples of its already decoded neighboring PBs in the current picture) and intra-prediction mode. It supports 35 intra-prediction modes: 33 angular modes, planar mode and DC mode. Reference substitution and smoothing are applied on reference samples in some cases [Sullivan et al., 2012; Lainema and Han, 2014].

- **Inter-prediction**: uses previously reconstructed pictures that are available in the Decoding Picture Buffer (DPB) as reference for Motion Compensation (MC), as well as Motion Vectors (MVs). Inter-prediction in performed on the PU level. MV is the resulting displacement between the area in the reference picture and the current PB. Regarding fractional reference picture samples interpolation, MVs are applied in quarter-sample accuracy with a 7/8-tap filter for luma inter-prediction, and eighth-sample accuracy with 4-tap filter for chroma inter-prediction. HEVC supports weighted prediction for both uni- and bi-prediction. It allows for two MV modes which are Advanced Motion Vector Prediction (AMVP) and merge mode [Sullivan et al., 2012; Bross et al., 2014].

- **Picture Reconstruction**: the reconstructed approximation of the residual samples resulting from de-quantization and inverse transform are then added to the intra- or inter-prediction samples to obtain the reconstructed CU.

- **In-loop Filters**: consist of a Deblocking Filter (DBF) followed by a Sampling Adaptative offset (SAO) that are applied to the reconstructed samples in the prediction loop before storing them in the DPB [Norkin et al., 2014]. The DBF is intended to reduce the blocking artifacts due to block-based coding. The SAO filter is intended to minimize the reconstruction error, enhance edge sharpness and suppress banding and ringing artifacts. While the DBF is only applied to the samples located at block boundaries, the SAO filter is applied adaptively to all samples satisfying certain conditions, e.g., based on gradient [Sullivan et al., 2012].

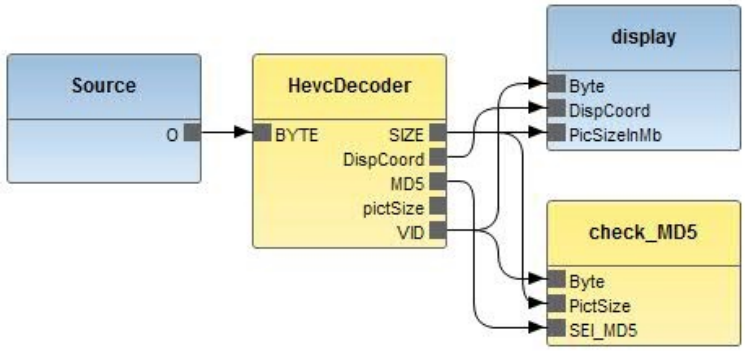


Figure 4.10: Top-level RVC FNL description of the HEVC decoder.

The HEVC Profile Definition A profile defines a set of coding tools or algorithms that can be used in generating a conforming bit-stream. Decoders conforming to a specific profile must support all features in that profile. Version 1 [Rec. H.265 , 04/13] of the HEVC standard defines 3 profiles (*Main*, *Main* 10 and *Main Still Picture*).

Main profile supports a bit depth of 8 bits per sample and 4 : 2 : 0 chroma sampling and employs the features described above.

Main 10 profile supports bit depth up to 10 bits with 4 : 2 : 0 chroma sampling.

Main Still Picture profile is a subset of of the Main profile for still image coding and thus interpicture prediction is not supported.

Version 2 [Rec. H.265 , 10/14] of HEVC adds 21 range extensions profiles, two scalable extensions profiles, and one multi-view profile. Version 3 [Rec. H.265 , 04/15] of HEVC adds the 3D Main profile.

4.4.1.2 The RVC-CAL HEVC Decoder

In parallel with the standardization process, the MPEG-RVC working group has standardized 3 video decoders using the RVC framework –MPEG–4 Visual, H.264/MPEG–4 AVC and HEVC– which are available in the Open RVC-CAL Applications (Orc–apps) open-source repository[1]. In the following, the RVC-CAL implementation of the HEVC decoder has been employed. Figure 4.10 shows a top-level RVC FNL description of the HEVC decoder. It encompasses 4 FUs: `Source` reads the video bit-stream from a file, `HEVCDecoder` is the main FU which is itself a hierarchical composition of actors, `Display` visualizes the decoded bit-stream and `Message Digest 5 (MD5)` verifies data integrity. An FNL description of the `HEVCDecoder` FU is shown in Figure 4.11, which is composed of 30 FUs and totals up 32 actors. Table 4.2 outlines the characteristics of the 10 FUs in the top of the hierarchy. Each FU is mapped to a common decoder functional block of Figure 4.9(b). Note that the structure of the decoder can be edited using Graphiti[2], a generic graph editor delivered as an Eclipse plug-in.

4.4.1.3 Test Sequences

The JCT–VC common test conditions define a set of configurations [Bossen, Oct. 2012] used in HEVC Test Model (HM) testing. These video sequences are compliant with different HM versions, encoded at various bit-rates and defined according to the picture size.

- Three configurations including All Intra (AI), Random Access (RA) and Low Delay (LD).
  - AI mode means that each image is coded as an Intra image without any prediction related to any other image. This mode can provide the best video quality but its compression efficiency is quite low.

[1] Orc–apps is available at:https://github.com/orcc/orc-apps
[2] Graphiti is available at: http://graphiti-editor.sf.net

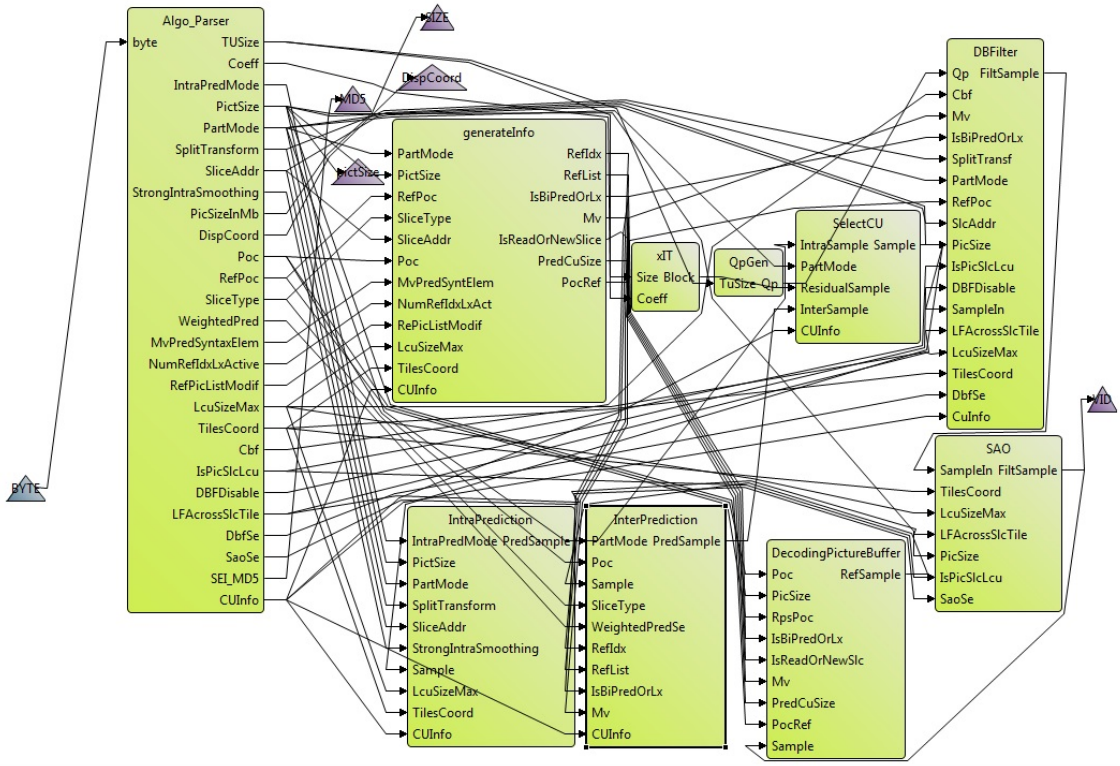


Figure 4.11: The RVC FNL description of the HEVC Decoder FU.

Table 4.2: Characteristics of the FUs of the `HEVCDecoder` FU.

FU	hier.	#FUs	#actors	Description
Algo_parser	no	n/a	1	corresponds to the entropy decoder.
generateInfo	yes	2	2	obtains the MVs, among other important information.
xIT	yes	26	21	implements the inverse transform and quantization.
QpGen	no	n/a	1	obtains the QP for each TU.
IntraPrediction	no	n/a	1	implements the intra-prediction.
InterPrediction	no	n/a	1	implements the MC.
DecodingPictureBuffer	no	n/a	1	corresponds to the DPB.
SelectCU	no	n/a	1	computes the picture reconstruction.
DBFilter	yes	2	2	corresponds to the DBF filter.
SAO	no	n/a	1	corresponds to the SAO filter.

  - RA mode means that inter image prediction tools are used inside a Group Of Pictures (GOP). This is the most common mode because it can provide the best quality vs. compression trade off.
  - LD mode is mainly used for videoconferencing services in order to warranty that the encoding delay will be compatible with an interactive service (that means using low complexity and robust tools that can work with small buffers).

- Six classes of test sequences.
  - A (4 sequences, 2560 × 1600, 30 and 60 Frames per Second (Fps))
  - B (5 sequences, 1920 × 1080, 24–60 Fps)
  - C (4 sequences, 832 × 480, 30–60 Fps)
  - D (5 sequences, 416 × 240, 30–60 Fps)
  - E (3 sequences, 1280 × 720, 60 Fps)
  - F (4 sequences, 832 × 480–1280 × 720, 20–50 Fps)

  Class A to E test sequences are camera captured content and class F contains screen content sequences. Please refer to Appendix B for video classes details[3].

- Four QPs including 22, 27, 32 and 37, where a QP of 37 produces very poor quality and high compression whereas a QP of 22 produces very low compression and high quality.

4.4.2 Optimization Metrics

In order to quantify the quality of our proposed design, two performance metrics are considered: time and area (Sub-section 2.2.3.1).
The time performance metrics are timing, throughput and latency. The standard metrics for timing are clock period and frequency. The maximum frequency of a FPGA design is determined by the delay of its longest path, referred to as the Critical Path (CP). Throughput is the amount of output samples per unit of time, measured in Samples per Second (Sps) or Fps. We can express the throughput as:

$$(throughput) = \frac{S}{T * C}. \tag{4.5}$$

where S is the total number of output samples, T is the clock period and C is the number of clock cycles required to compute the output samples. Latency is the time required, measured in units of time (milliseconds (ms)), to compute an output sample for a corresponding input sample.
Area is a measure of how many hardware resources are required to implement the design. Area can be measured as a number or a percent of available resources. For our experiments, the FPGA resource consumption (Sub-section 2.2.1.2) is given by the number of LUTs, FFs, slices, BRAMs and DSP blocks.

4.4.3 Experimental Setup

The system-level design flow for rapid prototyping of dataflow programs (Figure 4.3) is implemented using Orcc. Orcc is an open-source –under the BSD license– IDE based on Eclipse supporting source-to-source code transformation (Sub-section 3.5.2). The implementation flow (Appendix A) is summarized in Figure 4.12. The system-level specification is parsed into the design flow based on user applied directives. After source-to-source code transformation, the obtained *C*-HLS code is then synthesized and implemented using HLS and third-party tools. We use Vivado 2014.3 from Xilinx as the state-of-the-art HLS tool. The RTL output is implemented by Xilinx ISE 13.4 on the target FPGA platform Xilinx Virtex-7 (`XC7V2000T` package `FLG1925-1`). The Virtex-7 `XC7V2000T` is the the largest device currently available: it contains 6.8 billion transistors, providing customers access to 2 million logic cells. Area consumption and the CP delay are reported by ISE after P&R. ModelSim v10.1c is a package in Mentor Graphics and is used for logic simulation of HDLs. Experiments was done on a 2.93 GHz Intel Centrino Dual Core with 4 Go RAM running Windows 7 Professional.

[3] http://www.4ever-project.com/docs/files/4EVER_HEVC.pdf

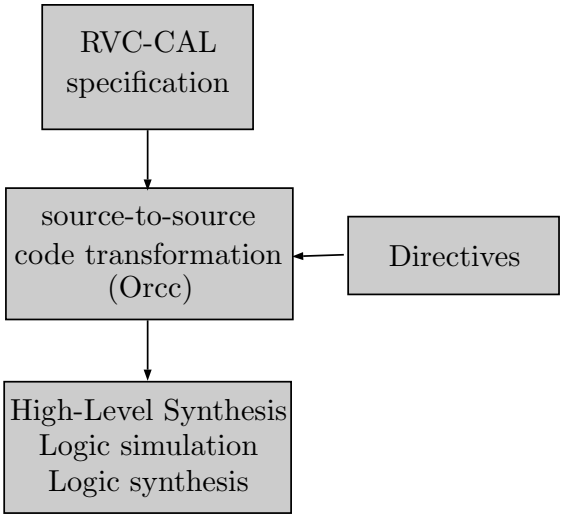


Figure 4.12: Implementation flow.

4.4.4 Experimental Results

In the case study, we are implementing the RVC-CAL HEVC decoder (Main Still Picture profile) into FPGA with the system-level design flow based on explicit streaming, that we denote the *Stream Design* during experiments. In this chapter, we show that a simulated and synthesized version of the RVC-CAL HEVC decoder (Main Still Picture profile) is rapidly obtained with promising preliminary results. We present simulation and synthesis results for different bit-streams encoded at different bit-rates (Figures 4.13(a), 4.13(b) and 4.13(d)):

- An AI Class-D HEVC video sequence `BQSquare` (416 × 240 image size, 60 Fps and QP 32).
- An AI and an RA Class-D HEVC video sequence `BlowingBubbles` (416 × 240 image size, 50 Fps and QP 32).
- An AI Class-D HEVC video sequence `RaceHorses` (416 × 240 image size, 30 Fps and QP 22).
- An AI Class-B (HD) HEVC video sequence `BasketballDrive` (1920 × 1080 image size, 50 Fps and QP 32).

The choice of such small video sequences resolution is advocated by the total amount of memory required for HEVC decoding. Indeed, most of the memory is required for the DPB that holds multiple pictures. That is why, the size of this buffer may be larger in HEVC for a given maximum picture size, and could exceed the available memory of any FPGA. Although, we tried to reduce this buffer by using low-resolution bit-streams, the DPB remains very memory consuming. A possible solution in this case would be the use of an Synchronous Dynamic RAM (SDRAM) which requires the development of a new Orcc back-end.

4.4.4.1 The Main Still Picture profile of HEVC case study

Simulation and synthesis results of the RVC-CAL HEVC decoder (Main Still Picture profile) are shown in Tables 4.3 to 4.6 for a stimulus frequency of 50MHz. The FIFO channels are bounded to 8192. As explained in Sub-section 4.3.5, the test infrastructure that we implemented inside Orcc allows tests at different levels including system-level and component-level. Hence, we simulated and synthesized some FUs of the RVC-CAL HEVC decoder (Main Still Picture profile) in standalone fashion allowing workload analysis. The high latency of the RVC-CAL HEVC decoder (Main Still Picture profile) using the *Stream Design* was expected since the `IntraPrediction` and the `SelectCU` FUs store a big amount of tokens before starting the processes. The throughput frequency can be far improved by exploiting Vivado HLS directives that will be exposed in the following. We notice that this is a pioneer simulated and synthesized

(a) BQSquare (416 × 240 image size, 60 Fps and QP 32).

(b) BlowingBubbles (416 × 240 image size, 50 Fps and QP 32).

(c) RaceHorses (416 × 240 image size, 30 Fps and QP 22).

(d) BasketballDrive (1920×1080 image size, 50 Fps and QP 32).

Figure 4.13: Decoded HEVC video sequences used in experiments.

Table 4.3: Time results for the RVC-CAL HEVC decoder (Main Still Picture profile) simulated by the *Stream Design* for 3 frames of the BQSquare video sequence at 50MHz.

	HEVCIntraDecoder
Latency (ms)	248.10
Sample Rate (Sps)	0.54×10^6
Throughput (Fps)	3.66

Table 4.4: Maximum operating frequency and area consumption for the SelectCU and IntraPrediction FUs of the RVC-CAL HEVC decoder (Main Still Picture profile) synthesized by the *Stream Design* for 3 frames of the BQSquare video sequence on a Xilinx Virtex 7 platform (XC7V2000T) at 50MHz.

	SelectCU	IntraPrediction
Maximum frequency (MHz)	161.499	70.630
Number of Slice Registers	1400	6753
Number of Slice LUTs	2473	15692
Number of Block RAM/FIFO	9	23

[1] Number of Slice Registers Available: 2443200 — [2] Number of Slice LUTs Available: 1221600 — [3] Number of Block RAM/FIFO Available: 1292

Table 4.5: Time results for the RVC-CAL HEVC decoder (Main Still Picture profile) simulated by the *Stream Design* for 3 frames of the BlowingBubbles video sequence at 50MHz.

	HEVCIntraDecoder
Latency (ms)	250
Sample Rate (Sps)	0.81×10^6
Throughput (Fps)	5

version of the RVC-CAL HEVC decoder (Main Still Picture profile) and the obtained results can be considered a starting point.

Table 4.6: Time results, maximum operating frequency and area consumption for the `xIT`, `Algo_Parser` and `IntraPrediction` FUs of the RVC-CAL HEVC decoder (Main Still Picture profile) synthesized by the *Stream Design* for 3 frames of the `BlowingBubbles` video sequence on a Xilinx Virtex 7 platform (`XC7V2000T`) at 50MHz.

	`xIT`	`Algo_Parser`	`IntraPrediction`
Latency (ms)	0,08602	0,76672	0,01986
Samples per second	4135831	2293599	2580387
Throughput (FPS)	27	15	17
Maximum frequency (MHZ)	86.417	84.863	70.630
Number of Slice Registers [1]	14052	26956	6753
Number of Slice LUTs [2]	218303	53106	15827
Number of Block RAM/FIFO [3]	50	2208	23

[1] Number of Slice Registers Available: 2443200 — [2] Number of Slice LUTs Available: 1221600 — [3] Number of Block RAM/FIFO Available: 1292

Table 4.7: Throughput, latency and maximum frequency results on different operating frequencies for the `BlowingBubbles` video sequence.

Frequency (MHz)	10	50	100	200	250
Latency (ms)	0.102	0.022	0.013	0.009	0.009
Throughput (Fps)	3	16	30	47	47
Maximum Frequency	69	68	108	162	129
Real Throughput (Fps)	22	22	32	37	23

4.4.4.2 The `IntraPrediction` FU Case Study

Table 4.7 shows the effect of operating frequency change on time performance of the `Intra Prediction` FU, where clock frequency is varied from 10MHz to 250MHz. Understanding of the effects of those variations is very important in the design of high performance digital system, because these significantly affect the CP delay or the maximum operating frequency. Our experiments affirmed that latency and throughput values are varying proportionally to the operating frequency. That is to say, if we increase frequency from F to αF ($\alpha < 5$), Latency is decreased from L to L/α and throughput is increased from T to αT.

4.4.4.3 DSE Through Optimizations Directives

In this section, we show how to perform fast DSE with the `SelectCU` FU of the RVC-CAL HEVC decoder as an example. To achieve this, we apply directive-based optimizations provided by Vivado HLS. Our goal is to optimize the RTL designs produced by our proposed system-level design flow to best adhere to designer-specified system goals such as area and/or performance requirements. Exploring designs using Vivado HLS directive-based optimizations does not require the *C*-HLS code to be altered. These optimizations are specified as directives using Tcl scripts, or can be embedded in the *C*-HLS source code at high-level. In our system-level design, directives are automatically generated from the system-level using the template-based pretty printer of Orcc. Then, a `directives.tcl` file containing the appropriate directives is automatically created for each actor and executed in batch mode. We have afterward to experiment with a variety of directives and determine through trial and error which directive will deliver an improvement.
A summary of Vivado HLS directives can be found in Appendix C, classified by optimization strategies (Area and/or time). For an in-depth explanation, please refer to [UG902 , v2015.2]. We examine in the following the most important directives that have been used in the optimization of the `SelectCU` FU of the RVC-CAL HEVC decoder (Table 4.8):

1. *Optimization* 1 *(Function Inlining)*: removes the function hierarchy and improves time performance by reducing function call overhead. A function is inlined using the `INLINE`

Table 4.8: Vivado HLS directive-based optimizations impact on the SelectCU FU of the RVC-CAL HEVC decoder (Main Still Picture profile) synthesized by the *Stream Design* for the RaceHorses video sequence on a Xilinx Virtex 7 platform (XC7V2000T) at 50MHz.

	Latency (ms)	Throughput (Fps)	LUTs [1]	Registers [2]
No opt.	0.00686	26	38505	51889
Opt. 1	0.00686	41	38522	51955
Opt. 2	0.00564	62	29376	51889
Opt. 3	0.00754	41	30643	53319
Opt. 4	0.00582	43	60114	56673
Opt. 5	0.00516	71	43755	54205
Opt. 6	0.00564	62	31322	51886

[1] Number of Slice LUTs Available: 1221600 — [2] Number of Slice Registers Available: 2443200

directive through the Tcl command set_directive_inline. The second line of Table 4.8 shows a 1.57× throughput improvement after *Function Inlining* of the SelectCU FU of the RVC-CAL HEVC decoder.

2. *Optimization* 2 *(Loop Unrolling)*: transforms for-loop by creating multiple independent operations rather than a single collection of operations, and enables all iterations to occur in parallel. A loop is unrolled using the UNROLL directive through the Tcl command set_directive_unroll. The third line of Table 4.8 shows a 3.38× throughput improvement and a 1.2× latency improvement after *Loop Unrolling* of the SelectCU FU of the RVC-CAL HEVC decoder.

3. *Optimization* 3 *(Array Mapping)*: combines multiple smaller arrays into a single large one to help reduce BRAM resources. An array is mapped using the MAP directive through the Tcl command set_directive_array_map. The fourth line of Table 4.8 shows a 1.57× throughput improvement and a decrease of 20% in area consumption after *Array Mapping* of the SelectCU FU of the RVC-CAL HEVC decoder.

4. *Optimization* 4 *(Array Partitioning)*: partitions large BRAM into smaller BRAMs or into individual registers, to improve access to data and remove BRAMs bottlenecks. An array is partitioned using the PARTITION directive through the Tcl command set_directive_array_partition. The fifth line of Table 4.8 shows a 1.65× improvement in throughput against more than 50% increase in resource consumption after *Array Partitioning* of the SelectCU FU of the RVC-CAL HEVC decoder.

5. *Optimization* 5 *(Array Reshaping)*: creates a single new array with fewer elements but with greater word-width and reduces the number of BRAM while still allowing the beneficial attributes of partitioning: parallel access to the data. An array is reshaped using the RESHAPE directive through the Tcl command set_directive_array_reshape. The sixth line of Table 4.8 shows a 2.73× improvement in throughput after *Array Reshaping* of the SelectCU FU of the RVC-CAL HEVC decoder.

6. *Optimization* 6 *(Binding)*: determines the effort level to use during the binding process (Section 2.5) and can be used to globally minimize the number of operations used. Binding is configured using the configuration config_bind. The seventh line of Table 4.8 shows a 2.38× throughput improvement and a decrease of 18% in area consumption after *Binding* of the SelectCU FU of the RVC-CAL HEVC decoder.

Ultimately, we come to several conclusions regarding the use of the Vivado HLS tool namely its ability (1) to bring hardware design at higher abstraction level, (2) to increase design productivity and (3) to allow fast DSE. DSE included partitioning a large RAM into smaller memories, inlining functions, unrolling loops and binding operations, etc, which would be tedious in HDL. In HDL designs, each scenario would likely cost an additional day of coding followed by then testbench modification to verify correct functionality. However, with Vivado HLS these changes took minutes and did not entail any major alteration of the source code. In other word, Vivado

HLS allows a designer to focus more on the algorithms themselves rather than the low-level implementation, which is error prone, difficult to modify and inflexible with future requirements.

4.5 Conclusion

The purpose of this chapter is to raise the level of abstraction in the design of embedded systems to the system-level. A novel design flow was proposed that enables an efficient hardware implementation of video processing applications described using the RVC-CAL dataflow programming language. Despite the huge advancements in HLS for FPGAs, designers are still required to have detailed knowledge about coding techniques and the targeted architecture to achieve efficient solutions. Moreover, the main downside of the HLS tools is the lack of the entire system consideration. As a remedy, in this chapter, we proposed a design flow that combines a dataflow compiler for generating *C*-based HLS descriptions from a dataflow description and a *C*-to-gate synthesizer for generating RTL descriptions. The challenge of implementing the communication channels of dataflow programs relying on MoC in FPGA is the minimization of the communication overhead. In this issue, we present in Chapter 5 a new interface synthesis approach that maps the large amounts of data that multimedia and image processing applications process, to shared memories on FPGA.

Bibliography

[Abid et al., 2013] M. Abid, K. Jerbi, M. Raulet, O. Deforges, and M. Abid. System level synthesis of dataflow programs: Hevc decoder case study. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pages 1–6, May 2013. 55, 75

[Bezati et al., 2011] 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. ix, 53, 54, 75

[Bossen, Oct. 2012] F. Bossen. Common hm test conditions and software reference configurations. *in Proc. 11th meeting*, Oct. 2012. 66, 75

[Bross et al., 2014] B. Bross, P. Helle, H. Lakshman, and K. Ugur. Inter-picture prediction in hevc. In V. Sze, M. Budagavi, and G. J. Sullivan, editors, *High Efficiency Video Coding (HEVC)*, Integrated Circuits and Systems, pages 113–140. Springer International Publishing, 2014. 65, 75

[Budagavi et al., 2014] M. Budagavi, A. Fuldseth, and G. Bjntegaard. Hevc transform and quantization. In V. Sze, M. Budagavi, and G. J. Sullivan, editors, *High Efficiency Video Coding (HEVC)*, Integrated Circuits and Systems, pages 141–169. Springer International Publishing, 2014. ISBN 978-3-319-06894-7. 65, 75

[Chen and Dömer, 2014] W. Chen and R. Dömer. *Out-of-order Parallel Discrete Event Simulation for Electronic System-level Design*. Springer International Publishing, 2014. ISBN 9783319087535. 62, 75

[Chi et al., 2012] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl. Parallel scalability and efficiency of hevc parallelization approaches. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1827–1838, Dec 2012. 65, 75

[Cooling and Hughes, 1989] J. Cooling and T. Hughes. The emergence of rapid prototyping as a real-time software development tool. In *Software Engineering for Real Time Systems, 1989., Second International Conference on*, pages 60–64, Sep 1989. 53, 75

[De Souza et al., 2014] D. De Souza, N. Roma, and L. Sousa. Opencl parallelization of the hevc de-quantization and inverse transform for heterogeneous platforms. In *Signal Processing Conference (EUSIPCO), 2014 Proceedings of the 22nd European*, pages 755–759, Sept 2014. 65, 75

[Janneck et al., 2008] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs: An mpeg-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 287–292, Oct 2008. 53, 75

[Jerbi, 2012] K. Jerbi. *High Level Hardware Synthesis of RVC Dataflow Programs*. Theses, INSA de Rennes, Nov 2012. 54, 59, 75

[Jerbi et al., 2012] K. Jerbi, M. Raulet, O. Deforges, and M. Abid. Automatic generation of synthesizable hardware implementation from high level rvc-cal description. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1597–1600, March 2012. ix, 54, 75

[Kim et al., 2012] I.-K. Kim, J. Min, T. Lee, W.-J. Han, and J. Park. Block partitioning structure in the hevc standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22 (12):1697–1706, Dec 2012. 65, 76

[Lainema and Han, 2014] J. Lainema and W.-J. Han. Intra-picture prediction in hevc. In V. Sze, M. Budagavi, and G. J. Sullivan, editors, *High Efficiency Video Coding (HEVC)*, Integrated Circuits and Systems, pages 91–112. Springer International Publishing, 2014. 65, 76

[Marpe et al., 2003] D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):620–636, July 2003. 65, 76

[Norkin et al., 2014] A. Norkin, C.-M. Fu, Y.-W. Huang, and S. Lei. In-loop filters in hevc. In V. Sze, M. Budagavi, and G. J. Sullivan, editors, *High Efficiency Video Coding (HEVC)*, Integrated Circuits and Systems, pages 171–208. Springer International Publishing, 2014. 65, 76

[Parr, 2004] T. J. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 224–233, New York, NY, USA, 2004. ACM. ISBN 1-58113-844-X. doi: 10.1145/988672.988703. URL http://doi.acm.org/10.1145/988672.988703. 54, 76

[Rec. H.265 , 04/13] Rec. H.265 (04/13). H.265 : High efficiency video coding, 2013. 66, 76

[Rec. H.265 , 04/15] Rec. H.265 (04/15). H.265 : High efficiency video coding, 20115. 66, 76

[Rec. H.265 , 10/14] Rec. H.265 (10/14). H.265 : High efficiency video coding, 2014. 66, 76

[Sjberg et al., 2012] R. Sjberg, Y. Chen, A. Fujibayashi, M. M. Hannuksela, J. Samuelsson, T. K. Tan, Y.-K. Wang, and S. Wenger. Overview of hevc high-level syntax and reference picture management. *IEEE Trans. Circuits Syst. Video Techn.*, 22(12):1858–1870, 2012. 65, 76

[Sullivan et al., 2012] G. Sullivan, J. Ohm, W.-J. Han, and T. Wiegand. Overview of the high efficiency video coding (hevc) standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1649–1668, Dec 2012. 63, 65, 76

[Sze and Marpe, 2014] V. Sze and D. Marpe. Entropy coding in hevc. In V. Sze, M. Budagavi, and G. J. Sullivan, editors, *High Efficiency Video Coding (HEVC)*, Integrated Circuits and Systems, pages 209–274. Springer International Publishing, 2014. 65, 76

[Sze et al., 2014] V. Sze, M. Budagavi, and G. Sullivan. *High Efficiency Video Coding (HEVC): Algorithms and Architectures*. Integrated Circuits and Systems. Springer International Publishing, 2014. 63, 76

[UG902 , v2015.2] UG902 (v2015.2). Vivado design suite user guide high-level synthesis, June 2015. 71, 76

[XIL, 2014] *Vivado Design Suite User Guide High-Level Synthesis UG902*. XILINX, October 2014. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_3/ug902-vivado-high-level-synthesis.pdf. 57, 76

[Yviquel et al., 2011] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet. Efficient multicore scheduling of dataflow process networks. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 198–203, Oct 2011. doi: 10.1109/SiPS.2011.6088974. 62, 76

5

Toward Optimized Hardware Implementation of RVC-based Video Decoders

> *This is how you do it: you sit down at the keyboard and you put one word after another until its done. It's that easy, and that hard.*
>
> Neil Gaiman

5.1 Introduction

This dissertation deals with the hardware implementation of dataflow-based video decoders that have been developed within the RVC framework. In Chapter 4, a fully automated design flow methodology was proposed. That is, from RVC-based video decoders specifications, a dataflow compilation infrastructure –Orcc– generates a *C*-based code, which is fed to a *C*-to-gate tool –Xilinx Vivado HLS– to generate synthesizable hardware implementation. The raise of the abstraction level to the system-level, by the use of RVC-CAL, presents several advantages in the design of video processing applications in terms of design productivity. Moreover, low-level implementation details are no longer taken into account since only the architecture of the dataflow program is considered at the system-level.
Nevertheless, the key issue in this design flow is the implementation and handling of FIFO channels, that may impact application performance in terms of both time and area. An appropriate design willing to achieve efficient hardware implementation of RVC-CAL dataflow programs need to minimize the unnecessary overhead introduced by FIFO accesses and scheduling of actions, involved by dynamic dataflow model on which the RVC-CAL language is built. That is, an actor is constituted by a set of actions that may fire according to the availability of tokens and to the fulfillment of guards conditions. Afterward, a scheduling policy is defined to select one action to fire.
The main contribution of this chapter is the enhancement of the implementation of the communication channels between components. The goal is to optimize performance metrics such as latency and throughput of dataflow-based video decoders. Therefore, the system-level design flow presented in Chapter 4 is enhanced according to optimized communications and scheduling. Section 5.2 highlights the limitations of the proposed *C*-HLS backend. The contribution of this chapter is detailed in Section 5.3. Section 5.4 presents the results achieved using two different RVC descriptions of the emerging HEVC standard: serial and parallel. We evince by comparing test results how communication and scheduling overhead is reduced by an optimized communication mechanism and scheduling policies.

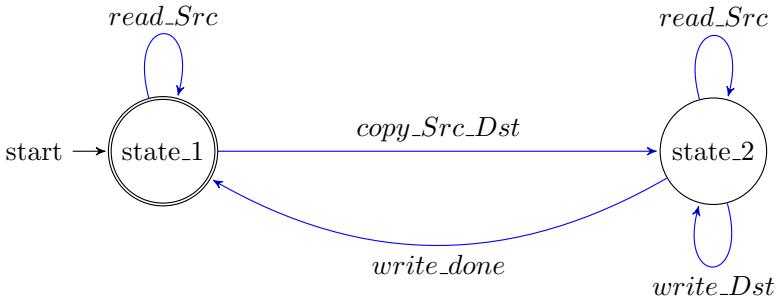


Figure 5.1: The FSM of the `Transpose32 × 32` actor.

5.2 Issues with Explicit Streaming

The first contribution of this dissertation, as described in Chapter 4, involves the design of an automated tool-chain within the RVC framework that allows fast hardware generation from system-level dataflow programs. In other words, the first primary goal is the correct and efficient implementation of DPN-based programs onto FPGAs which was a challenging task. However, in this proposed implementation, the overhead of the action scheduling and communication via channels is considerable. The reason is the strong expressive power of the DPN MoC behind dynamic dataflow programs, which states inter alia that both actors production and consumption are not known a priori, i.e. actors can receive and send data at any rate, as explained in Sub-section 3.4.2. Moreover, the M2M tokens transformation used to cache data locally in the proposed solution with streams, as explained in Paragraph 4.3.2.3, modifies heavily the structure of the actor by adding internal buffers for each input port. Besides, for each input port, it adds an action for consuming input tokens from a stream and storing them into a local input buffer. Then, it adds an action that performs the core computation, possibly fills a local output buffer for each output port. Finally, it adds an action that writes the tokens toward the output stream, as well as an FSM for coordinating these actions. In the worst-case scenario, if an actor consumes n tokens from its input port and produces m tokens on its output port, we need to add at least $n + m + 1$ steps to fire an action. We illustrate this problem by considering the following example. That is, the multi-rate token production and consumption is a recurring phenomenon when dealing with blocks of pixels in video decoder, such as the transposition of 32×32 block presented in Listing 5.1 that reads 1024 tokens from its input port `Src` and copies them in a new order to its output port `Dst`.

```
1  package org.sc29.wg11.mpegh.part2.main.IT;
2
3  actor Transpose32x32 () int(size=16) Src
4                          ==>
5                          int(size=16) Dst
6                          :
7
8    action Src:[ src ] repeat 1024 ==> Dst:[ dst ] repeat 1024
9      var
10     List(type:int(size=16), size=16) dst
11   do
12           Dst := [ src[ 32 * column + row ]:
                 for int row in 0 .. 31, for int column in 0 .. 31 ];
13   end
14 end
```

Listing 5.1: Transposition of a 32×32 block in RVC-CAL

Considering the proposed solution with explicit streaming introduced in Chapter 4, the RVC-CAL description of Listing 5.1 is translated into a *C*-HLS code whose action scheduler is illustrated in Figure 5.1. As presented in Sub-section 3.4.2, an action firing is an indivisible quantum of computation composed of three ordered and indivisible steps as illustrated in Figure 5.1:

1. **Reading**: the procedure `read_Src` consumes input tokens in order from the input stream and stores them into an internal input buffer. This procedure is executed 1024×.

2. **Processing**: the procedure `copy_Src_Dst` performs processing of tokens as defined in its RVC-CAL description and fills an internal output buffer. Note that data processing is not performed in order. This procedure is executed 1×.

3. **Writing**: the procedure `write_Dst` consumes tokens from the internal output buffer and writes them toward the output stream. This procedure is executed 1024×.

Although the proposed solution with streams respects the DPN MoC, it requires additional copies between the streams and the internal buffers. Additionally, the consumption and production of tokens are not done in parallel due to the sequential model of firing actions in an actor, which can be a significant bottleneck for performances.
Hence, the second goal is the minimization of the communication and scheduling overhead. In DPN MoC, the fact that actors are connected by passing tokens along channels, means that communication is separated from the computation as evoked in Section 2.6. This is beneficial for the system optimization since it enables us to focus on each concern independently – computation and communication. Hence, we propose a novel approach for an optimized communication and scheduling refinement to improve real-time performance constraints for video decoders.

5.3 Interface Synthesis Optimization

After explanation of how to model video decoders and how to synthesize them from a higher level of abstraction in Chapter 4, this section addresses performance bottleneck introduced by both the scheduling policy and the communication mechanism. This optimization has been integrated in the design flow we have introduced in Chapter 4.

5.3.1 Shared-Memory Circular Buffer

The main bottleneck of the previous proposed solution with explicit streaming described in Chapter 4 mainly lies on streaming interface which gives rise to communication overhead. To tackle this problem, we enhanced the *C*-HLS backend by using implicit streaming rather than explicit one. Thereby, interface ports are declared as external one-dimensional arrays so allowing access to the external shared-memory as depicted in Listing 5.2 for the `Select` actor.

```
1 #define FIFO_SIZE 512
2 typedef signed char i8;
3 //Input FIFOS
4 extern i8 tab_A[FIFO_SIZE];
5 extern i8 tab_B[FIFO_SIZE];
6 extern bool tab_S[FIFO_SIZE];
7 //Output FIFOS
8 extern i8 tab_Output[FIFO_SIZE];
```

Listing 5.2: The *C* declaration of the interface ports of the actor `Select` using implicit streaming.

Interface ports are defined by specifying the type, the variable name and the array size. For example, a 8-bit integer type is defined and used to create an array called `tab_A` of size 512 in Listing 5.2. The reason for using arrays of a fixed-size is that memory is limited in physical systems and FIFOs size should be effectively specified. Estimation of minimal required FIFOs size is impossible for dynamic dataflow models with timing and data dependencies. Therefore, the general practice is that the FIFO size is initially guessed as the maximum communication rate within the application and later increased if insufficient. For hardware designs, FIFOs size setting out is valuable in that it impacts the resource usage, functionality and performance. Having FIFOs that are too large consumes resources unnecessarily, which may increase the cost of an implementation of a DPN specification. Having FIFOs that are too small may causes the system to deadlock. Conversely, the resulting implementation may be slowed unnecessarily. The issue of FIFO size optimization was addressed in [Brunet et al., 2013; Ab Rahman, 2014] based

on critical path analysis of RVC-CAL dataflow programs.
In a fixed-size FIFO, writing and reading operations are concurrently executed. To mediate actors' communication with respect to the DPN semantic rules, bounded FIFOs are implemented as circular buffers allocated in shared-memory. The data structure of a circular buffer consists of a memory array and read and write indexes to array elements as presented in Listing 5.3 and Figure 5.2. Both indexes are stored in unsigned integer variables.

```
1 struct FIFO {
2   tokenType tab_A[FIFO_SIZE];
3   unsigned int rIdx_A;
4   unsigned int wIdx_A;
5 };
```

Listing 5.3: Data structure of FIFO channel `A` of the actor `Select`.

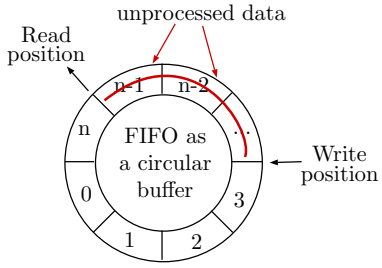


Figure 5.2: Conceptual view of a circular buffer.

Using circular buffers to implement FIFOs requires efficient index management. In the following, writing process and reading process in the circular buffer are respectively expressed as producer and consumer. As shown in Figure 5.3, the producer and consumer actors communicate according to independent policies for reading and writing data to the FIFO while adhering to the rule that only the producer actor modifies the write index and only the consumer actor modifies the read index. Since an action firing is an indivisible quantum of computation, indexes are

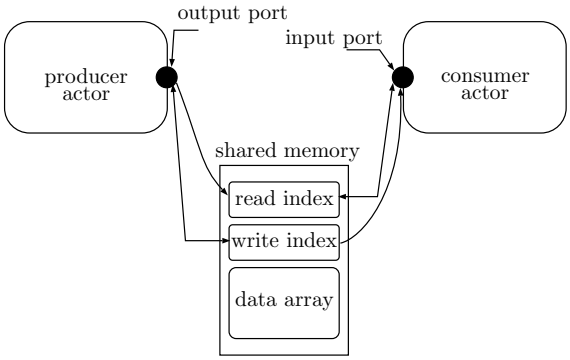


Figure 5.3: A shared-memory circular buffer used to mediate communication between actors with respect to the DPN semantics.

incremented only once at the end of the action while preserving the DPN semantics. In other words, the producer actor cannot access the FIFO involved by a reading process until the read index is updated, and the consumer actor cannot access the FIFO involved by a writing process until the write index is updated, as well.
Whereas writing and reading increase the indexes infinitely until the overflow of the variables, fixed-size FIFO requires to use the modulo operation of `FIFO_SIZE` to roll back to zeroth location once the end of the array is reached. Since computing the modulo is costly on hardware, it has been translated into a bit-and operation by forcing the size of the buffer to a power-of-two. For the purpose to make the state of the read or write index visible by the producer or consumer respectively, writing a copy of the read or write index respectively to a read or write index in

shared memory is required. In other words, read and write indexes are transmitted at the end of the action on external interface ports declared as one-dimensional arrays of size 1 (Listing 5.4).

```
1 extern unsigned int writeIdx_A[1];
2 extern unsigned int readIdx_A[1];
3
4 readIdx_A[0] = rIdx_A;
5 writeIdx_A[0] = wIdx_A;
```

Listing 5.4: Read and write indexes are stored on shared-memory one-dimensional arrays of size 1 for the actor `Select`.

Thereby, each producer/consumer actor manages its own index for writing/reading, but can get access to both indexes' states.
The difference between the code example of Listing 5.2 and that of Listing 4.2 lies on the communication mechanism. When using arrays instead of streams as interface ports in the generated *C*-HLS code, we get rid of adding internal buffers to cache data locally, and data are pulled/pushed directly from/to the kth location of the array. In other words, accesses to the FIFOs (i.e. store, load and peek operations) are carried out by accessing directly to the content of the arrays as they are implemented as shared-memory and the additional copies to the internal buffers are removed:

- Write operation is achieved by accessing the buffer according to write index updated just once at the end of the action (Listing 5.5).

```
1 i8 v;
2 tab_Output[wIdx_Output & FIFO_SIZE] = v;
3 wIdx_Output = wIdx_Output + 1;
```

Listing 5.5: Write operation with implicit streaming.

- Read operation is achieved by accessing the buffer according to read index updated just once at the end of the action (Listing 5.6).

```
1 i8 v;
2 v = tab_A[rIdx_A & FIFO_SIZE];
3 rIdx_A = rIdx_A + 1;
```

Listing 5.6: Read operation with implicit streaming.

- Peek operation is achieved by accessing the buffer directly without the update of the read index, whereas peek operation with explicit streaming is carried out through the buffering mechanism (Listing 5.7).

```
1 bool s;
2 s = tab_S[rIdx_S & FIFO_SIZE];
```

Listing 5.7: Peek operation with implicit streaming.

5.3.2 Scheduling Optimization

The overhead caused by the scheduling policies of the solution with explicit streaming can lead to performance bottleneck. To overcome this issue, the action scheduler is optimized, as shown in Listing 5.8 for the actor `Select`, so as to evaluate the firing rules that determine the fireability of an action that way:

- The input pattern: every time there is data read, there has to be a check for the amount of tokens required in the input channel (Line 2 in Listing 5.8).
- The peek pattern: every time there is a guard condition on the actor's internal state and/or a peek into the input tokens' value, there has to be a check for the validity of the condition (Line 4 in Listing 5.8).

- The output pattern: every time there is a data write, there has to be a check for full buffer, i.e. the availability of enough space in the output channel (Line 5 in Listing 5.8).

The unsigned difference (e.g. (`wIdx_A - rIdx_A`) for the actor `Select`) yields the number of tokens placed in the shared-memory circular buffer and not yet retrieved, and thus indicates the state of the buffer (empty or full) as highlighted in Listing 5.8. Using this methodology, the firing rules of Equations (3.1) and (3.2) are implemented in Equations (5.1) to (5.6) by using efficient indexes management of shared-memory circular buffer as follow:

$$P_{1,1} = [writeIdx\_A[0] - rIdx\_A >= 1] \tag{5.1}$$

$$P_{1,3} = [writeIdx\_S[0] - rIdx\_S >= 1] \tag{5.2}$$

$$G_{1,3} = [S\_buffer[readIndex\_S] = true] \tag{5.3}$$

$$P_{2,2} = [writeIdx_B[0] - rIdx_B >= 1] \tag{5.4}$$

$$P_{2,3} = [writeIdx_2S[0] - rIdx_S >= 1] \tag{5.5}$$

$$G_{2,3} = [S\_buffer[readIndex\_S] = false] \tag{5.6}$$

```
1  void Select_scheduler() {
2  if (writeIdx_A[0] - rIdx_A >= 1  &&
3  writeIdx_S[0] - rIdx_S >= 1  &&
4  isSchedulable_select_a()&&
5  (FIFO_SIZE - wIdx_output + readIdx_output[0] >= 1)){
6      Select_select_a();
7      }
8  else if (writeIdx_B[0] - rIdx_B >= 1  &&
9  writeIdx_S[0] - rIdx_S >= 1  &&
10 isSchedulable_select_b()) &&
11 (FIFO_SIZE - wIdx_output + readIdx_output[0] >= 1)){
12     Select_select_b();
13     }
14 }
```

Listing 5.8: The optimized action scheduler of the actor `Select`.

In case of success, the evaluation of the firing rule is followed by the firing of the associated action (Line 6 in Listing 5.8).
The implementation of the action scheduler of the solution with explicit streaming and that with implicit streaming, namely in case of multi-rate communication (e.g. Listing 5.1), shows that the reading is done in parallel, in contrast to the serial reading resulting from the M2M transformation. Besides, the parallel production of tokens uses non-blocking writes, which means that the action fires only if the output FIFO has a space, thus eliminating the need to create a new action just for the writing of tokens to FIFO as the M2M transformation does. Further, the solution with implicit streaming requires neither additional copies between the streams and the internal buffers as tokens are directly pulled/pushed from/to the shared-memory circular buffer, nor an FSM for coordinating these steps as illustrated in 5.1 for explicit streaming. To summarize, the three first steps of action firing (reading, processing and writing) are merged together, thus reducing the number of instructions to fire an action. Moreover, the FIFO indexes are updated after the action processing, thus letting the other actors use newly produced data in parallel.

5.3.3 Synthesis of Arrays

As described in Sub-section 4.3.3, the HLS stage with Vivado HLS performs two distinct types of synthesis upon the design: algorithm synthesis and interface synthesis. However, interface ports with the solution with implicit streaming are implemented in the RTL as an `ap_memory` interface. This type of interface port is intended to communicate with a standard block RAM resource within the FPGA with data, address, Chip-Enable (CE) and Write-Enable (WE) ports

as illustrated in Figure 5.4 for the `Select` actor. In order to ensure array variables are targeted at the correct memory type, we define the FPGA-specific hardware (single-port RAM) using the set directive resource command of Vivado HLS for each array, e.g. `set_directive_resource -core RAM_1P "Select_select_a" tab_A`.

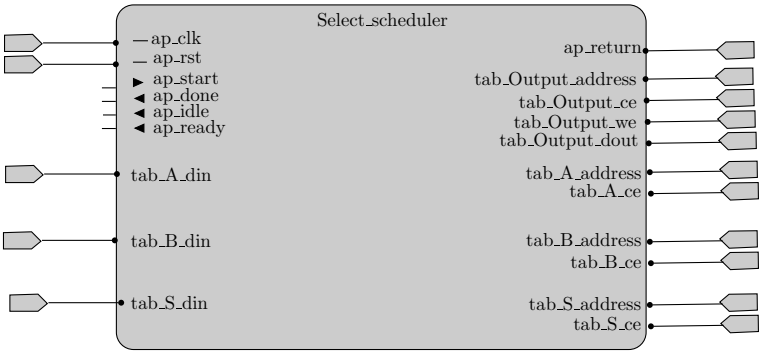


Figure 5.4: The corresponding RTL implementation of the interface ports of the actor `Select` using implicit streaming.

As each actor writing into and reading from a shared memory RAM knows each own write and read index respectively (e.g. `rIdx_A` and `wIdx_A` for the actor `Select`), each corresponding local variable is implemented in RTL as internal signal. Whereas, write and read one-dimensional arrays (e.g. `readIdx_A[1]` and `writeIdx_A[1]` for the actor `Select`) where counts of the number of tokens written to and read from the circular buffer are stored in shared-memory, are implemented in the RTL as an ap_memory interface with data, address, CE and WE ports as well. The following timing diagram of Figure 5.5 describes the temporal behavior of the `Select` actor using implicit streaming.

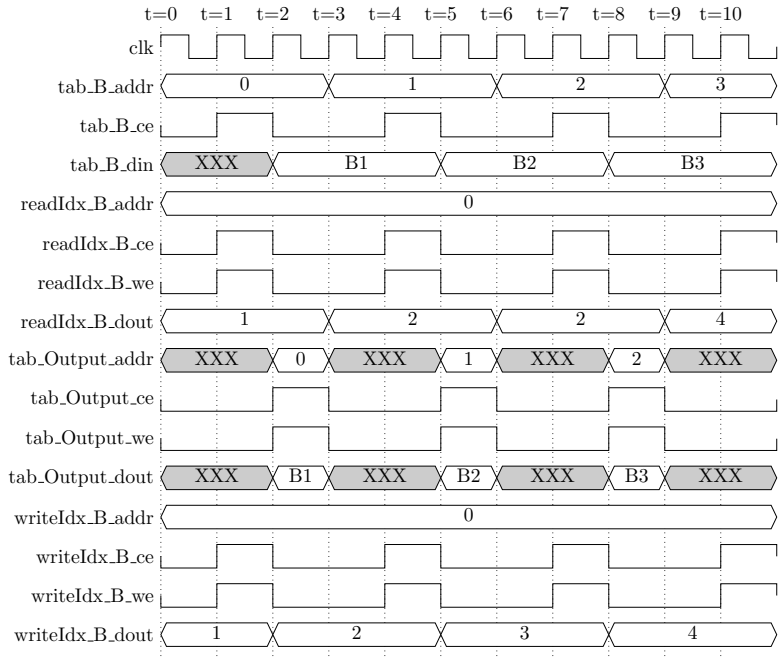


Figure 5.5: Timing behavior of `ap_memory` interface ports of the actor `Select`.

5.3.4 System-Level Integration

RAM inference RAM inference is the process of synthesizing a memory block (RAM) from a HDL program. By employing the template-based pretty printer of Orcc (Sub-section 3.5.2, we write a VHDL code that properly declares and defines a dual-port RAM, using separate read and

write ports (since two memory accesses can occur simultaneously) as illustrated in Figure 5.6. Hence, the bit-width of the elements, the address bus width as well as the size of the memory are automatically configured (Listing 5.9).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ram_tab is
    generic(
            dwidth     : integer := 32;
            awidth     : integer := closestLog_2(fifoSize)  ;
            mem_size   : integer :=  fifoSize
           );
    port (
            addr0     : in std_logic_vector(awidth-1 downto 0);
            ce0       : in std_logic;
            q0        : out std_logic_vector(dwidth-1 downto 0);
            addr1     : in std_logic_vector(awidth-1 downto 0);
            ce1       : in std_logic;
            d1        : in std_logic_vector(dwidth-1 downto 0);
            we1       : in std_logic;
            clk       : in std_logic
         );
end entity;
```

Listing 5.9: RAM inference using pretty printing techniques in Orcc.

This step is crucial to explicitly associate the RTL descriptions of each actor with the dedicated memory blocks, which we detail in the following.

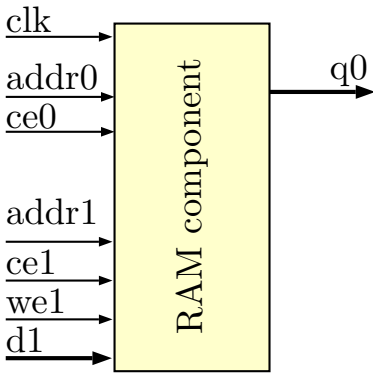


Figure 5.6: The RAM component implementation.

Through a straightforward translation of the RVC-CAL network into VHDL, the set of actors components and block RAMs are instantiated and every dataflow connection is replaced with the appropriate handshaking signals. As explained in Sub-section 4.3.4, we do not need to schedule the actors in hardware since all actors can run in parallel pursuant to the flow of tokens. Moreover, the size of each RAM is defined automatically according to the design in order to avoid deadlock. Indeed, the RAM component is instantiated through inference by adding a generic clause for the corresponding size (Listing 5.10).

```
connection.ramName  : ram_tab
    generic map (
                dwidth     => connection.fifoType.sizeInBits,
                awidth     => closestLog_2(connection.safeSize)  ,
                mem_size   => connection.safeSize)
    port map (
              clk => top_ap_clk,
              addr0 => top_connection.ramName_address0,
```

```
9               ce0 => top_connection.ramName_ce0,
10              q0 => top_connection.ramName_q0,
11              addr1 => top_connection.ramName_address1,
12              ce1 => top_connection.ramName_ce1,
13              we1 => top_connection.ramName_we1,
14              d1 => top_connection.ramName_d1
15      );
```

Listing 5.10: RAM instantiation using pretty printing techniques in Orcc.

5.3.5 Test Infrastructure

Using pretty printing techniques, test benches are automatically generated in Orcc depending on the need to evaluate the performance at the system-level or at the actor-level or at the action-level. The test bench compares the outputs of the generated VHDL code with reference values. These reference values correspond to the traces of the FIFOs generated using the *C* backend, i.e. the tokens flowing within each FIFO buffer of the RVC-CAL actor-network are recorded as FIFOs traces. In the following, we detail the elaborated test infrastructure inside the proposed design flow:

1. In order to simulate an actor in a standalone fashion, a test infrastructure is carried out. Actors that write/read into/from the memory buffer of an input/output port, respectively, are added on either side of the current actor. As well, a test bench is generated for each actor that accepts stimulus text files of each input and output port of the actor, i.e. the FIFO traces are used to execute and analyze each RVC-CAL actor individually.

2. In order to identify the inefficient areas of the code which require optimization at the actor-level, an action debug feature is added for each actor. This new feature provides a Gantt-chart for each actor by recording the start and end of each action execution in order to reveal the dependencies between actions.

3. In order to simulate the whole design, a test bench file is generated for the network. A script, when executed, generates the hardware components of the whole network with a single click. This enables to evaluate the global performance of the system.

5.4 Experimental Results

The goal of Section 5.4.1 is to firstly show the achieved improvements with the system-level design flow based on implicit streaming, that we denote the *RAM Design* during experiments, on the RVC-CAL HEVC decoder (Main Still Picture profile). Then, we demonstrate in Section 5.4.2 a simulated version of the RVC-CAL HEVC decoder (Main profile) followed by performance bottleneck analysis and prospective Vivado HLS directive-based optimization. Finally, in Sections 5.4.3 and 5.4.4, we show how to exploit parallelism to improve performance. As case study, we target 2 versions of an RVC-CAL HEVC video decoder. On the one hand, the standardized serial RVC-CAL HEVC video decoder (Figure 4.11) denoted the *Ref Design*. On the other hand, the parallel RVC-CAL HEVC video decoder denoted the *YUV Design*. The *YUV Design* is also available in the Orc–apps open-source repository. As explained in Section 5.3.5, the test infrastructure that we implemented inside Orcc allows tests at different levels including system-level, actor-level and action-level.

5.4.1 The Main Still Picture profile of HEVC case study

In this section, we implement the RVC-CAL HEVC decoder (Main Still Picture profile) into FPGA with the *RAM Design* (explained in Section 5.3) to demonstrate performance improvement compared to the *Stream Design* (explained in Section 4.3).

Stream Design* vs. *RAM Design We proceed by the logic simulation and synthesis of the RVC-CAL HEVC decoder by the *RAM Design* as depicted in Tables 5.1 to 5.4. Compared to Tables 4.3 to 4.6, simulation results show a throughput improvement with a speed-up factor of

Table 5.1: Time results for the RVC-CAL HEVC decoder (Main Still Picture profile) simulated by the *RAM Design* for 3 frames of the `BQSquare` video sequence at 50MHz.

	`HEVCIntraDecoder`
Latency (ms)	64.83
Sample Rate (Sps)	2.71×10^6
Throughput (Fps)	18.11

Table 5.2: Maximum operating frequency and area consumption for the `SelectCU` and `IntraPrediction` FUs of the RVC-CAL HEVC decoder (Main Still Picture profile) synthesized by the *RAM Design* for 3 frames of the `BQSquare` video sequence on a Xilinx Virtex 7 platform (`XC7V2000T`) at 50MHz.

	`SelectCU`	`IntraPrediction`
Maximum frequency (MHz)	161.160	70.630
Number of Slice Registers	1348	6443
Number of Slice LUTs	2494	15219
Number of Block RAM/FIFO	4	21

[1] Number of Slice Registers Available: 2443200 — [2] Number of Slice LUTs Available: 1221600 — [3] Number of Block RAM/FIFO Available: 1292

Table 5.3: Time results for the RVC-CAL HEVC decoder (Main Still Picture profile) simulated by the *RAM Design* for 3 frames of the `BlowingBubbles` video sequence at 50MHz.

	`HEVCIntraDecoder`
Latency (ms)	58
Sample Rate (Sps)	3.66×10^6
Throughput (Fps)	24

5.2× as well as a latency improvement with a speed-up factor of 3.8× with the *RAM Design* compared to the *Stream Design*. Indeed, the results largely depend on the replacement of the copies between the streams and the internal buffers in the *Stream Design*, by a shared memory communication in the optimized *RAM Design*. Besides, in the *RAM Design*, the three first steps of action firing (Reading, processing and writing) are merged together, thus reducing the number of instructions to implement an action. However, this number of instructions reduction remains negligible ahead the number of lines of code of the `Algo_Parser` FU due to the fine grained communication rate, which explains the low values of the throughput of this FU (Column 3 of Table 5.4). Moreover, the FIFO indexes are updated after the action processing, thus letting the other actors using newly produced data in parallel. That is why the optimized design leads to the achievement of good performance. There is however between 10% and 50% improvement in resource consumption on average.

5.4.2 The Main profile of HEVC case study

According to the previous results, the *RAM Design* has lead to performance improvement compared to the *Stream Design* since we enhanced communication and scheduling refinement in hardware generation of dataflow-based video decoders. We notice however that the design is far from meeting the designer-specified latency and throughput goals. So, to identify the inefficient areas of the code which require optimization, we took the entire RVC-CAL HEVC decoder (Main profile) and compiled it using the *RAM Design*. Then, we evaluated the hardware implementation for each actor independently in a standalone simulation (Table 5.5). Finally, we constructed Gantt-chart for each actor by recording the start and end of each action execution and revealing the dependencies between actions (Figure 5.8). The remainder of this section discuss also the optimizations that we could perform to achieve the highest throughput, lowest latency FPGA

Table 5.4: Time results, maximum operating frequency and area consumption for the `xIT`, `Algo_Parser` and `IntraPrediction` FUs of the RVC-CAL HEVC decoder (Main Still Picture profile) synthesized by the *RAM Design* for 3 frames of the `BlowingBubbles` video sequence on a Xilinx Virtex 7 platform (`XC7V2000T`) at 50MHz.

	`xIT`	`Algo_Parser`	`IntraPrediction`
Latency (ms)	0.03454	0.76384	0.01386
Sample Rate (Sps)	8470414	3495591	4668145
Throughput (FPS)	56	23	31
Maximum frequency (MHZ)	72.386	84.864	70.630
Number of Slice Registers [1]	278201	26824	6488
Number of Slice LUTs [2]	170212	56287	15210
Number of Block RAM/FIFO [3]	96	2200	21

[1] Number of Slice Registers Available: 2443200 — [2] Number of Slice LUTs Available: 1221600 — [3] Number of Block RAM/FIFO Available: 1292

Table 5.5: Simulation results of the HEVC decoder (Main profile) for 10 frames of the `BlowingBubbles` video sequence on a Xilinx Virtex 7 platform (`XC7V2000T`) at 50MHz..

Actors	**Latency (ms)**	**Throughput (Sps)**	**Throughput (Fps)**
`IntraPrediction`	0.014	1563565.68	10.44
`InterPrediction`	1,823	2173110,003	14,51
`Algo_Parser`	0,789	3118969,44	20,82
`xIT`	0,942	9597782,66	64,08
`SelectCu`	0,002	10000006,68	66,77
DBF	14,209	5982586,74	39,95
SAO	13,645	5805193,43	38,76
DPB	14,529	10000006,68	66,77
`HEVCInterDecoder`	61,67	1529111,75	10,21

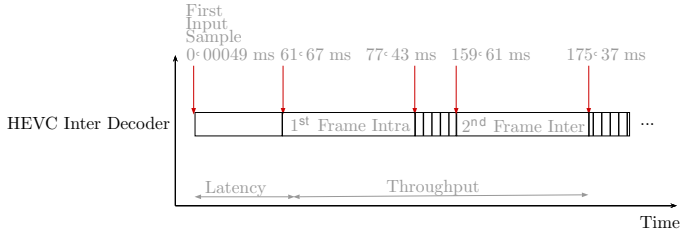


Figure 5.7: Gantt-chart of the HEVC Inter Decoder.

implementation by applying Vivado HLS directive-based optimizations (Section 4.4.4.3) after bottlenecks analysis.

5.4.2.1 Throughput Analysis

The overall throughput bottleneck is primarily due to the wait time between the last sample of a picture till the first sample of the next picture is decoded, as illustrated in Figure 5.7. This is evidenced by the fact that the DBF and the SAO filters are implemented with picture-based processing, which needs the whole picture samples to be stored before filter process. This wait time is taken into consideration in the overall throughput computation in Equation (4.5). That is why a shorter wait time between decoded pictures would increase throughput.
Moreover, the results in Table 5.5 clearly show that the throughput evaluation of each actor independently is not equitably balanced. This difference can be partially explained by the fact that the HEVC decoder is still being under development, especially concerning the complexity of the actors. That is to say, the `Algo_Parser`, the `interPrediction` and the `IntraPrediction` FUs are by far the most complex actors in the network. Another explanation is the difference of

granularity of the decomposition between the components of the decoder: the inverse transform is hierarchical and designed with 21 actors (the `xIT`), while most of the other components are designed with an unique actor.
Table 5.5 also shows that the `IntraPrediction` FU is a throughput bottleneck, which dictates the overall throughput. Indeed, the `IntraPrediction` is done with neighboring blocks in the same picture (spatial prediction). That is why the algorithm performs mostly iterative operations across windows in the picture. In order to achieve the required performance, we carefully analyze each stage in the algorithm. The resulting Gantt-chart of the `IntraPrediction` FU is shown in Figure 5.8(a), where the action `getSamples_launch` reads input samples and the action `sendSamples_launch` writes output samples. The Gantt-chart shows that the throughput bottleneck resides in the action `sendSamples_launch`. There are two issues that limit the throughput in this action:

- The action body contains 3 nested for-loops: By default loops are kept rolled in Vivado HLS and one copy of the loop body is synthesized by using the same hardware resources and re-used for each iteration. This ensures each iteration of the loop is sequentially executed. That is why the for-loop should be unrolled to allow all operations to occur in parallel and increase throughput.
- The action body contains also 3 "reading from" arrays `lumaComp`, `chComp_u` and `chComp_v`. Arrays are by default implemented as BRAMs in hardware which only has a maximum of two data ports. This can limit the throughput. The throughput can be improved by partitioning these arrays (BRAMs resources) into multiple smaller arrays (individual registers), effectively increasing the number of ports.

5.4.2.2 Latency Analysis

At first, the important overall latency of the `HEVCInterDecoder` is due to the fact that the DPB, the DBF and the SAO FUs store a big amount of tokens before starting the process (Table 5.5). In order to further locate the latency bottleneck, we carefully analyze each action of these actors through the Gantt-charts shwon in Figures 5.8(b) to 5.8(d) respectively. These Gantt-charts take into account:

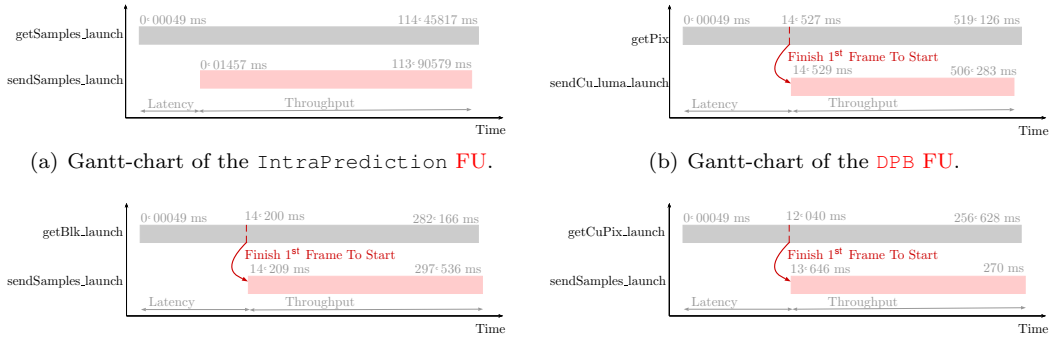
- The execution time of the actions that read input samples `getPix`, `getBlk_launch` and `getCuPix_launch`, of these actors respectively.
- The execution time of the actions that write output samples `sendCu_luma_launch`, `sendSamples_launch` and `sendSamples_launch`, of these actors respectively.
- The dependencies between the action that read input samples and the action that write output sample.

Figures 5.8(b) to 5.8(d) show that there are strong data dependencies between the action that reads input samples and the action that writes output samples, which causes a latency bottleneck. The problem lies in the fact that the action that writes output sample requires all 149760 tokens (equivalent to 416×240 YUV picture size) to be ready before the computation can start. There are two issues that limit the latency in the bodies of the actions `getPix`, `getBlk_launch` and `getCuPix_launch` :

- Each action body contains 2 nested for-loops. Typically, it requires additional clock cycles to move between rolled nested loops. It requires one clock cycle to move from an outer loop to an inner loop and from an inner loop to an outer loop. So, the fewer the number of transitions between loops, the less the time a design will take to start. Unrolling or flattening a loop hierarchy allows loops to operate in parallel, which in turn decreases latency.
- Each action body contains also an n-D ($n \in \{3, 4\}$) array `pictureBuffer`. Access to arrays in Vivado HLS can create performance bottleneck since they are implemented as BRAMs. Array partitioning can be used therefore to improve latency. Moreover the array `pictureBuffer` creates data dependencies between loops inside actions. That is to say the actions that write output samples and access the `pictureBuffer` array for read cannot begin until the actions that read input samples have finished all write accesses to

the array `pictureBuffer`. Data dependencies often prevent maximal parallelism and minimal latency. The solution is to try ensure the actions that read input samples are performed as early as possible.

- The action body of `getCuPix_launch` contains in addition functions calls. We can reduce function call overhead by removing all function hierarchy (inline) in order to improve latency.



(a) Gantt-chart of the `IntraPrediction` FU.

(b) Gantt-chart of the `DPB` FU.

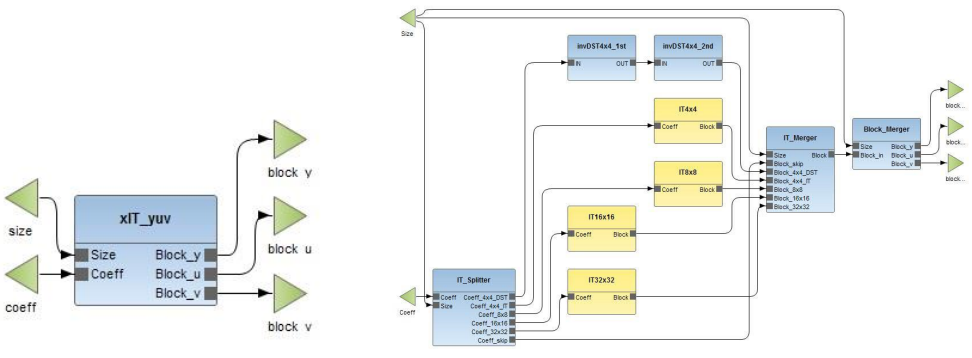
(c) Gantt-chart of the `DBF` FU.

(d) Gantt-chart of the `SAO` FU.

Figure 5.8: Latency bottleneck analysis using Gantt-chart.

5.4.3 Task Parallelism Optimization

In the previous section, we explained how to analyze throughput and latency bottleneck in order to apply Vivado HLS directive-based optimizations. In order to further improve the system performance, optimizations of the dataflow program can be performed at the system-level. Indeed, designers have the possibility to increase the level of parallelism by using *refactoring* techniques of actors or actions. *Refactoring* of an actor/action essentially means splitting, replicating, or modifying its computational elements such that an increase in parallelism is obtained. At the level of actors, the `xIT` actor illustrated in Figure 5.9(a) is partitioned into the sub-network illustrated in Figure 5.9(b). Each actor of the new sub-network performs a different set of operations, which typically requires the final merging of results as shown by the actor `Block_Merger`. Results in a real-time 50fps 1080p HEVC video sequence are shown in Table 5.6. Compared to the original serial implementation of the `xIT` actor, the partitioned implementation achieves a throughput improvement of roughly $4,23\times$ and a latency increase by $1,17\times$, since additional explicit parallelism is now exposed by partitioning the actor into several ones.



(a) (b)

Figure 5.9: *Refactoring* of the `xIT` actor.

Table 5.6: Latency and sample rate improvement achieved when *refactoring* the `xIT` actor for the `BasketballDrive` video sequence at 50MHz.

	Serial `xIT`	Parallel `xIT`
Latency (ms)	$1,17$	$0,93$
Sample Rate (Sps)	$1,20 \times 10^6$	$5,10 \times 10^6$

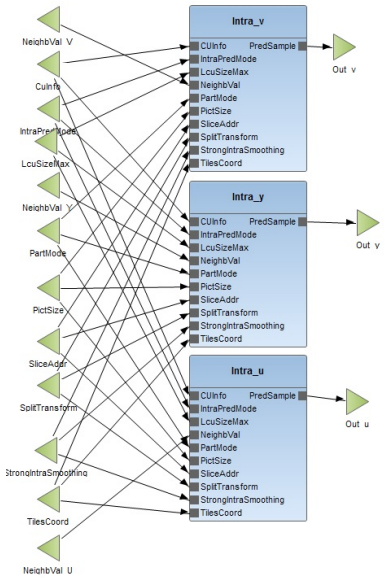


Figure 5.10: Example of the YUV-parallel split of the `IntraPrediction` FU.

5.4.4 Data parallelism optimization

5.4.4.1 YUV-Parallel RVC-CAL HEVC decoder

YUV-Parallel RVC-CAL HEVC decoder is also composed by 10 FUs as illustrated in Figure 4.11. However, the so-called parallelism is due to the fact that the decoding process is split into three parallel processes according to the color space components Y, U, and V. The YUV splitting is applied to all FUs except the `Algo_Parser` and the `xIT` FUs since the bit-stream of the three layers is merged in the input video stream and it would be difficult to separate it. An example of the YUV-parallel split of the `IntraPrediction` FU is illustrated in Figure 5.10. A simple splitting of the YUV components can increase the theoretical performance by 33%.

5.4.4.2 *Ref Design* vs. *YUV Design*

System-level simulation By approaching the system at the system-level by using the RVC-CAL dataflow language, the designer is better able to optimize not only the inter-actors communication but also to increase the level of parallelism. Table 5.7 compares the *Ref Design* to the *YUV Design*. Difference between the parallel and serial version of the RVC-CAL HEVC decoder is the content of token channels. For the parallel version, Y, U and V tokens are processed in respective channel, whereas for the serial version, Y, U and V tokens are combined in one

Table 5.7: Time results comparison between the *Ref Design* and the *YUV Design* both simulated by the *RAM Design* for 5 frames of the `BlowingBubbles` video sequence at 50MHz.

	Ref Design	*YUV Design*
Latency (ms)	$64,83$	$3,98$
Sample Rate (Sps)	$2,71 \times 10^6$	$3,25 \times 10^6$
Throughput (Fps)	$18,11$	$21,71$

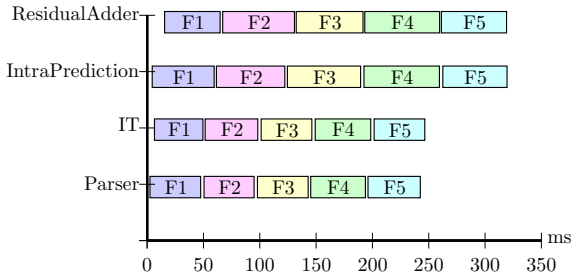


Figure 5.11: Graphical representation of the actors behavior of the YUV design simulation: The frame decoding start and end times are recorded for each actor during the system simulation for an image sequence of 5 frames.

Table 5.8: Time results of the *YUV* design (Main Still Picture Profile) synthesized by the RAM design for an image sequence of 5 frames and a 16384 FIFO size.

	`ALgo_Parser`	`xIT`	`IntraPrediction`	`SelectCu`
Latency (ms)	$0,76$	$0,01$	$0,005$	$1,54$
Sample Rate (Sps)	$3,09 \times 10^6$	$8,46 \times 10^6$	$4,5 \times 10^6$	$3,28 \times 10^6$

channel sequentially. Results show that the parallel version of the decoder introduces $16,58\%$ increase in term of throughput and $93,86\%$ increase in term of latency over the serial one. Those results show that the parallel decoder seems to be a better starting point when targeting hardware implementations. In order to track the actors behavior in the system, the top-level test bench allows us to build Gantt diagram by recording the decoding start and end times of the current frame for each actor. Figure 5.11 presents Gantt diagram of the YUV-parallel RVC-CAL HEVC decoder which supports the parallel and concurrent aspects when dealing with RVC-CAL dataflow programs.

Actor-level simulation We evaluated also the hardware implementation for each actor of the YUV-parallel HEVC decoder independently in a standalone simulation in Table 5.8. This enables us to know the maximum throughput and the minimum latency reached by each actor independently. Moreover, this enables us to know the bottleneck actors. Hence, The `SelectCu` FU is clearly the bottleneck actor in the *YUV Design* since it is computationally complex. That is why we do not meet the 33 % theoretical improvement compared to the *RVC Design* since the `SelectCu` FU slows down the *YUV Design*.

Action-level simulation In order to improve latency and throughput, we carefully analyze the algorithm of the `SelectCu` FU to identify actions that are latency and throughput bottleneck. To do so, we make use of the action debug feature explained in Section 5.3.5. There are two issues that limit the latency and the throughput in the `SelectCu` FU, including:

- The action bodies contain for-loops: By default loops are kept rolled in Vivado HLS, and one copy of the loop body is synthesized by using the same hardware resources and re-used for each iteration. This ensures each iteration of the loop is sequentially executed. That is why the for-loop should be unrolled to allow all operations to occur in parallel and increase throughput.

- The action bodies contain in addition functions calls. We can reduce function call overhead by removing all function hierarchy (inline) in order to improve the latency.

To circumvent these issues, the *set_directive_inline* and *set_directive_unroll* commands of vivado HLS are applied on function calls and loops respectively. Improvement results are showed in Table 5.9, with a gain of 32% in latency and throughput in the optimized `SelectCu` FU.

Table 5.9: Vivado HLS Directives are applied to the `SelectCu` FU.

	`SelectCu`	Optimized `SelectCu`
Latency (ms)	1.54	1.04
Sample Rate (Sps)	$3,28 \times 10^6$	$4,91 \times 10^6$

5.4.5 Comparison with Other Works

5.4.5.1 System-level hardware synthesis versus hand-coded HDL of the HEVC decoder

The purpose of this section is to compare the hardware synthesis from a dataflow-based HEVC decoder with the *RAM Design* against a low-level HEVC architecture without SAO designed by Tikekar et al. Tikekar et al. [2014]. In Tikekar et al. [2014], results for an ASIC test chip are presented. The chip achieves 249×10^6 Sps decoding throughput for luma-only (3840×2160)@30fps at 200 MHZ, which is equivalent to 41×10^6 Sps at 50 MHZ. This result shows that the manual VHDL HEVC implementation of Tikekar et al. [2014] is faster when compared to the automatically generated HDL with the proposed *RAM Design*. Indeed, the proposed design flow achieves only 1.5×10^6 Sps decoding throughput for (416×240)@50fps at 50 MHZ targeting the Virtex–7. One of the reasons why the implementation proposed in Tikekar et al. [2014] is faster is that the optimizations are applied at very low-level. However, the most important advantage with the proposed design flow (combining the Orcc compiler and the Vivado HLS tool) consists on considering the system-level of abstraction, which allows better complexity management, shorter development time and rapid system exploration. Moreover, it reduces the time-to-market and improves the RTL quality and the final performance of the design.

5.4.5.2 Comparison with other alternative HLS for RVC-CAL

In relation to similar works in literature, we mentioned in Section 3.5 that the hardware synthesis from RVC-CAL programs has gone through various evolution from the OpenDF framework (CAL2HDL [Janneck et al., 2008]) to the Orcc framework (ORC2HDL [Bezati et al., 2011] and Xronos [Bezati et al., 2013]). In my conference article [Abid et al., 2013], the simulation results of the hardware implementation of the MPEG-4 Simple Profile (SP) decoder generated with the proposed system-level design flow (Orcc + Vivado HLS) with explicit streaming (Chapter 4) are compared to those obtained with the Xronos tool. Figure 5.12 shows the MPEG-4 Part 2 SP decoder as described within RVC. It is essentially composed of 4 main FUs: the parser, a luminance component (Y) processing path, two chrominance component (U, V) processing paths and a merger. Each of the path is composed by its texture decoding engine as well as its motion compensation engine.

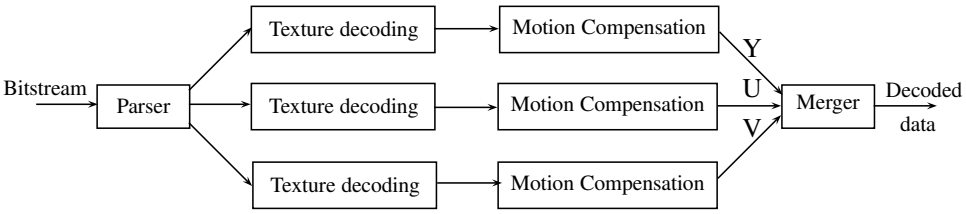


Figure 5.12: RVC-CAL description of the MPEG-4 SP decoder.

The simulated performance values are given in Table 5.10 for a stimulus frequency of 50 MHZ. Here, a Motion-MPEG stream consists of five QCIF images (176×144 pixels) has been used to obtain latency and throughput values. Considering the comparison in Table 5.10, our proposed system-level design flow (Orcc + Vivado HLS) with explicit streaming is found to be more efficient in terms of latency an less efficient in terms of throughput. Indeed, the design synthesized by Vivado HLS has a speed up factor of 1.6 in terms of latency compared to Xronos. However, Xronos has a speed up factor of 1.8 in terms of throughput compared to the design synthesized by Vivado HLS. Here it should be noted that system-level design flow (Orcc + Vivado HLS) with explicit streaming is employed. In view to maximize the total throughput, the system-level design flow (Orcc + Vivado HLS) with implicit streaming proposed in this chapter would give

considerable results. Moreover, not all advantages of Vivado HLS have been exploited. Unlike Xronos , Vivado HLS offers directive-based optimizations that could be exploited to maximize throughput.
Besides, we demonstrated a pioneer hardware implementation of the RVC-CAL HEVC decoder with our proposed system-level design flow (Orcc + Vivado HLS) on Xilinx 7 Series FPGAs. However, none of the related work on HLS for RVC-CAL has demonstrated a hardware implementation of the RVC-CAL HEVC, and none has demonstrated the hardware synthesis on Xilinx 7 Series FPGAs.

Table 5.10: MPEG-4 SP timing results.

	Orcc + Vivado HLS	Xronos
Latency (ms)	0, 158	0, 258
Throughput (Fps)	125	232

5.5 Conclusion

In this chapter, we proposed an enhanced hardware implementation of dataflow programs within the RVC framework. When dealing with the hardware synthesis of dataflow programs in the proposed design flow of Chapter 4, interface synthesis is the most important issue. Interface synthesis can be defined as the realization of communication between components via hardware resources and thus could very well be the bottleneck to meet performance requirements. Moreover, the resulting hardware should preserve the dataflow MoC semantics. A difference against previous works is that our approach enhanced the hardware implementation of the communication channels in dataflow programs by using a shared memory (RAM) that behaves as a circular buffer instead of a FIFO with additional storage elements. The experiments were performed on the dataflow-based implementation of the HEVC decoder, which has been recently implemented. Simulation results showed that the proposed implementation with RAM blocks has increased throughput and reduced latency compared to the state-of-the-art implementation proposed in Chapter 4. Another key feature of our proposed design flow is the ability to provide performance improvement by refactoring of the RVC-CAL programs. In other words, modeling applications at the system- level in the RVC framework eases system-level and component-level optimization in favor of hardware implementation disregarding hardware details.

Bibliography

[Ab Rahman, 2014] A. A. H. B. Ab Rahman. *Optimizing Dataflow Programs for Hardware Synthesis*. PhD thesis, STI, Lausanne, 2014. 79, 95

[Abid et al., 2013] M. Abid, K. Jerbi, M. Raulet, O. Deforges, and M. Abid. System level synthesis of dataflow programs: Hevc decoder case study. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pages 1–6, May 2013. 92, 95

[Bezati et al., 2011] 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. 92, 95

[Bezati et al., 2013] E. Bezati, M. Mattavelli, and J. Janneck. High-level synthesis of dataflow programs for signal processing systems. In *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*, pages 750–754, Sept 2013. 92, 95

[Brunet et al., 2013] S. Brunet, M. Mattavelli, and J. Janneck. Buffer optimization based on critical path analysis of a dataflow program design. In *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pages 1384–1387, May 2013. 79, 95

[Janneck et al., 2008] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs: An mpeg-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 287–292, Oct 2008. 92, 95

[Tikekar et al., 2014] M. Tikekar, C. Huang, C. Juvekar, V. Sze, and A. P. Chandrakasan. A 249-mpixel/s HEVC video-decoder chip for 4k ultra-hd applications. *J. Solid-State Circuits*, 49 (1):61–72, 2014. 92, 95

6 Conclusion

In the context of an increased interest for dataflow programming for designing embedded systems, this thesis discusses system-level hardware synthesis of dataflow programs with HEVC as study use case. Furthermore, this thesis addresses the problem of communication and scheduling overhead caused by FIFO-based communication channels in dynamic dataflow programs. In the following, we summarize the main contributions of this work while mentioning strengths and limitations of our work at the end of each paragraph.

The first contribution of this research work is an original technique that raises the level of abstraction to the system-level in order to obtain RTL descriptions from dataflow descriptions. First, we design image decompression algorithms using an actor oriented language under the RVC framework (RVC-CAL). Once the design is achieved, we use a dataflow compilation infrastructure called Orcc to generate a *C*-based code. Afterward, a Xilinx HLS tool called Vivado HLS is used for an automatic generation of synthesizable hardware implementation. The proposed system-level approach for generating hardware description from dataflow programs involves the implementation of a new *C*-HLS back-end of Orcc that considers the DPN-based model semantics and which is synthesizable by the newly Xilinx Vivado HLS tool as detailed in Chapter 4. That is, the functionality of the Vivado HLS tool was enhanced so it supports the entire system. Moreover, the methodology used to adapt such tool to the constraints of DPN-based model mainly the FIFO management was explained. The outcome of this contribution is threefold. On the one hand, the essential aim of our proposed rapid prototyping methodology is to alleviate the complexity gap problem and speed the time-to-market by quickly producing RTL descriptions from system-level dataflow programs. That is, our proposed design flow for hardware generation from system level is fully automated whatever the complexity of the application, which leads to gain in development time compared with manual approach. On the other hand, our development environment, known as Orcc, offers the possibility to translate the same system-level RVC-based descriptions of video decoders into both hardware (*C*-HLS back-end) and software (*C* back-end) equivalent descriptions intended for various platforms (FPGAs, MPSoC, respectively). Finally, by using Vivado HLS, we can take advantage of Vivado HLS optimization directives which enables easy and fast DSE to find the most-optimal implementation.

Chapter 5 emphasizes the second contribution of this research work. It consists of the enhancement of the communication and scheduling mechanisms to minimize the unnecessary overhead introduced by FIFO accesses and scheduling of actions, involved by dynamic dataflow model –on which the RVC-CAL language is built. This contribution answers the following question: what is the best way to connect components designed with RVC-CAL at the system-level while preserving dataflow programming features notably parallelism and concurrency? The main bottleneck of the previous proposed solution described in Chapter 4 lies mainly on streaming interface based on explicit streaming. To tackle this problem, we enhanced the *C*-HLS backend by using

implicit streaming rather than explicit one. In other words, our approach enhanced the hardware implementation of the communication channels in dataflow programs by using a shared memory (RAM) that behaves as a circular buffer with efficient index management instead of a FIFO with additional storage elements. Using dual-port block RAMs instead of FIFO buffers has the advantage to enable parallelism by allowing access to a common storage array through two independent access ports. Consequently, scheduling may take advantage of this by reading from one port and writing via another. Moreover, using dual-port block RAMs results in higher throughput and lower latency of RVC-CAL dataflow programs for hardware implementation. The major obstacle we faced is the fact that these kind of low level interface optimizations require advanced hardware domain expertise and that was challenging for us as software developers.

Once interface synthesis optimization has been achieved, the third contribution of this research work has been devoted to investigate system-level optimizations for increasing the efficiency and performance of the hardware design. Hence, by exploiting all the features of CAL and dynamic dataflow MoCs, we can optimize the high-level code by implementing task and data-level parallelism using the refactoring of the RVC-CAL programs for DSE. Refactoring is the process of changing the internal structure of a program through merging and splitting, while preserving its behaviour. RVC-CAL specifications have the advantage to expose all the parallelism possibilities intrinsic to video decoder applications. For this reason, refactoring involves application task partitioning and data partitioning. Some advantages of refactoring are as follows. First, the design space can be explored effectively for multiple criteria, including throughput and latency for real-time decoding. Second, refactoring makes the code easier to change and is very efficient in promoting better design and reuse, thus increasing design productivity.

The last contribution of this research work is to prove the applicability of our proposed rapid prototyping methodology for dataflow programs and to apply all the optimization methodologies evoked above. With the standardization of the new HEVC decoder, an RVC-CAL implementation of the HEVC decoder is also available as part of the standard. For that, an implementation of the most recent video decoder HEVC via our proposed rapid prototyping methodology has been demonstrated throughput the thesis. In Chapter 4, we have shown that a simulated hardware implementation of the RVC-CAL HEVC decoder is rapidly obtained with promising preliminary results. Although our proposed method compared to manual HDL approaches appears to be less efficient in terms of area consumption and performance, we effectively achieved a pioneer simulated hardware implementation of the most recent video coding standard HEVC with a very short time to market, while overcoming high computational complexity intrinsic to HEVC. Moreover, in Chapter 5, we have shown that when investigating optimization strategies whether through task and data-parallelism implementation or through Vivado HLS directives-based optimizations, we could improve performance and area metrics of the hardware implementation of the HEVC decoder. The obtained results after applying the proposed optimization strategies motivate us to investigate future research directions.

Based on the results and conclusions drawn from each of our contributions, we detail perspectives that would be interesting to explore.
A first important area of research is to ensure that the HEVC decoder achieves real-time performance for 4K Ultra-High Definition (UHD) video using our proposed system-level design flow. This can be achieved by exploring the design space for criteria such as throughput, latency and resource by applying optimization strategies proposed in this thesis, and by considering a solution for the DPB, the major memory bottleneck of the HEVC decoder. Additionally, we could use block RAM for large sized memories and distributed RAM for small sized memories, in order to avoid wastage of the space in RAM. Finally, we could minimize resource with buffer size optimization strategies.

A second important area of research is hardware/software codesign. Codesign stands for the joint design of software and hardware components from a single-application description. Since our development environment Orcc offers the possibility to automatically generate both hardware and software implementations from a unique RVC-CAL dataflow description, a codesign flow could perform the mapping of the components onto the available computing resources. That is, some FUs of the HEVC decoder are naturally suitable for software processing while

others are naturally suitable for hardware processing. For example the entropy decoder FU in the HEVC decoder executes a completely sequential algorithm with little computation and therefore is suitable for operating on processor. Whereas the inverse quantization and transform FU instantiates a large number of small actors with highly computational algorithms and thus adapted to hardware processing as they require high performance. The shared memory architecture (RAM) proposed in this thesis is the starting point that enables hardware-software codesign.

Another important area of research include using the frame-based implementation of the RVC-CAL HEVC decoder, to benefit from the increased data parallelism of this design. A pioneer dataflow description of the frame-based HEVC decoder has been recently developed by the Image team of the IETR laboratory. Thanks to the modularity of dataflow modeling, the frame-based parallelization is a duplication of the whole decoding process in order to enable the decoding of frames in parallel. Theoretically, the frame-based approach will improve the performance of the existing HEVC decoder by increasing the cadence through parallel frames decoding.

using a high-level synthesis tool enabled them to easily explore alternative architectures, which often led to more efficient implementations.

Part III

APPENDIX

A

System-Level Design Flow: Tutorial

A.1 A User Guide for the *C*-HLS Backend: Steps and Requirements

The purpose of this tutorial is to provide directions to obtain RTL descriptions from a system-level dataflow description using Orcc and Xilinx Vivado HLS tool.

Step 1 Write a simple RVC-CAL program

Step 2 Compile RVC-CAL program to *C*-HLS and VHDL codes using Orcc's *C*-HLS backend.

Step 3 Compile the *C*-HLS code to VHDL with Vivado HLS

Step 4 Synthesize the VHDL from Orcc and Vivado HLS to an FPGA using Xilinx ISE.

This compilation sequence is supported by the following software tools:

- Eclipse (with Orcc installed[1])
- Xilinx ISE Design toolset
- Xilinx Vivado HLS

A.2 Write a Very Simple Network

A.2.1 Setup a New Orcc Projet

First of all you need to create a new Orcc project in Eclipse (File >New >Other...). You can name it `AddOrcc` (Figures A.1(a) and A.1(b)). Then you can make a new package (File >New >Package) (Figure A.1(c))

A.2.2 Implements Actors

You should implement each actor using the RVC-CAL language. For each actor, you have to create a standard file in the right package. For example, the first actor to implement have to be written in the file `Add.cal` and so on (Figure A.2).

[1] A user guide for Orcc installation is available at: http://orcc.sourceforge.net/getting-started/install-orcc/

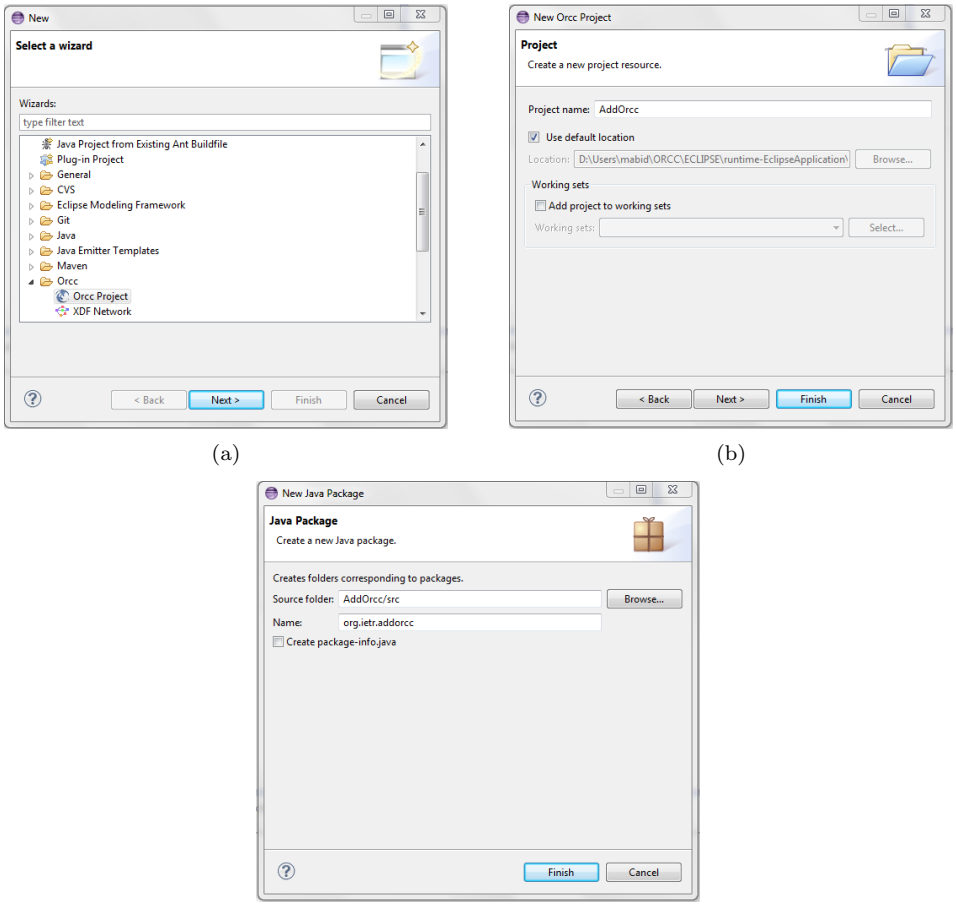


Figure A.1: Step 1: How to create a new Orcc project.

Add.cal

```
package org.ietr.addorcc;

actor Add () int(size=8) Input1, int(size=8) Input2 ==> int(size=8) Output:
  action Input1: [a], Input2: [b] ==> Output: [a + b]
  do
    println("Actor Add is adding " + a + " and " + b);
  end
end
```

(a)

Data.cal

```
package org.ietr.addorcc;

unit Data :

  int INPUT_SIZE = 5;

  int SRC1[INPUT_SIZE] = [
    1, 2,  3,  4,  5
  ];

  int SRC2[INPUT_SIZE] = [
    1, 2,  3,  4,  5
  ];

end
```

(b)

Actor1.cal

```
package org.ietr.addorcc;

import org.ietr.addorcc.Data.SRC1;
import org.ietr.addorcc.Data.INPUT_SIZE;

actor Actor1 () ==> int(size=8) source1 :

  int i := 0;

  sendData: action ==> source1:[ Out ]
  guard
    i < INPUT_SIZE
  var
    uint(size=8) Out
  do
    Out := SRC1[i];
    i := i+1;
  end
end
```

(c)

Actor2.cal

```
package org.ietr.addorcc;

import org.ietr.addorcc.Data.SRC2;
import org.ietr.addorcc.Data.INPUT_SIZE;

actor Actor2 () ==> int(size=8) source2 :

  int i := 0;

  sendData: action ==> source2:[ Out ]
  guard
    i < INPUT_SIZE
  var
    uint(size=8) Out
  do
    Out := SRC2[i];
    i := i+1;
  end
end
```

(d)

Actor3.cal

```
package org.ietr.addorcc;

actor Actor3 () int(size=8) result ==> :
  action result:[a] ==>
  do
    println("The result is " + a);
  end
end
```

(e)

Figure A.2: Step 1: Source code of actors to implement in RVC-CAL.

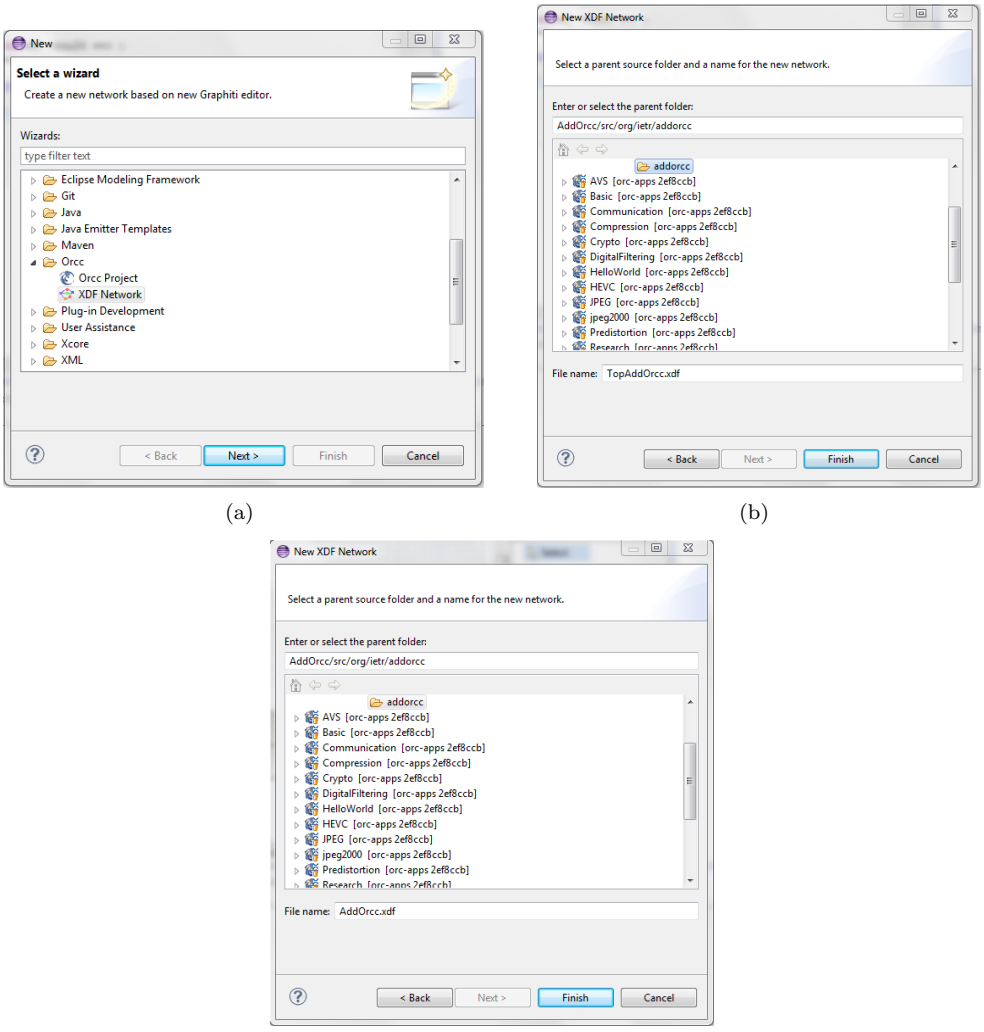


Figure A.3: Step 1: How to build an Orcc network.

A.2.3 Build the Orcc Network

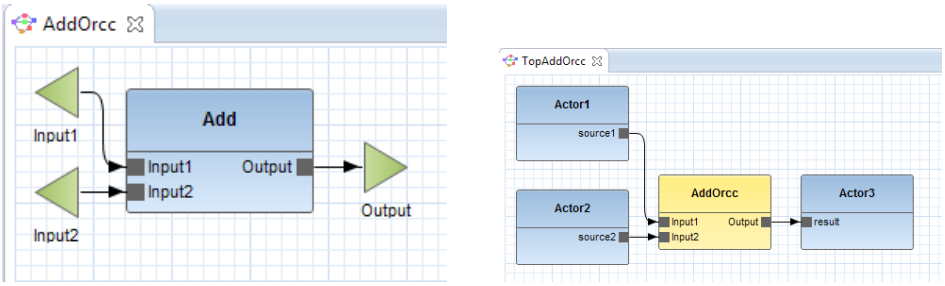
You have to create new Orcc Networks (File >New >Other...) (Figure A.3):

1. a top Network `TopAddOrcc.xdf`
2. a sub-network `AddOrcc.xdf`

Try to reproduce the following top network and sub-network using the palette on the right of the network editor (Figure A.4):

- Instance `Actor1` is associated to the actor `Actor1.cal`
- Instance `Actor2` is associated to the actor `Actor2.cal`
- Instance `Actor3` is associated to the actor `Actor3.cal`
- Instance `AddOrcc` is associated to the subnetwork `AddOrcc.xdf`

As for video decoder programming, `Actor1` and `Actor 2` import data and `Actor3` prints the results.



(a) (b)

Figure A.4: Step 1: How to build an Orcc network.

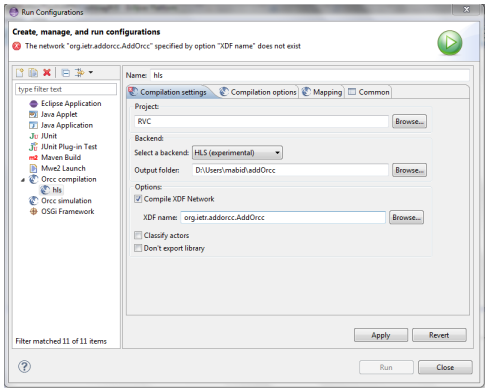


Figure A.5: Step 2: How to run an XDF network using the C-HLS backend.

A.3 Compile RVC-CAL Program Using the C-HLS Backend

- Run >Run Configurations (Figure A.5)
- In Compilation settings >Backend, select HLS (Experimental).
- Specify an output folder: `D:/addOrcc` for example. A folder named `HLSBackend` will be created automatically.
- In Options >compile XDF network, set the XDF name as `org.ietr.addorcc.AddOrcc` (i.e. the sub-network):

If Orcc generation of the C-HLS Backend succeeded, the compilation console shows the output of Figure A.6: Check the output folder `HLSBackend` and see what has been generated. The actual C-HLS backend of Orcc generates several files namely (Figure A.7):

- `batchCommand` Folder:
  1. `Command.bat` to generate HDL files using Vivado HLS for the Whole network.

```
12:39:01 : Lists actors...
12:39:01 : Instantiating...
12:39:01 : Flattening...
12:39:02 : Transforming actors...
12:39:02 : Printing children...
12:39:02 : Done in 0.87s
12:39:02 : Printing network... Done
12:39:03 : Printing network testbench... Done
12:39:03 : Printing network VHDL Top... Done
12:39:03 : Printing batch command... Done
12:39:04 : Orcc backend done.
```

Figure A.6: Step 2: Compilation console output.

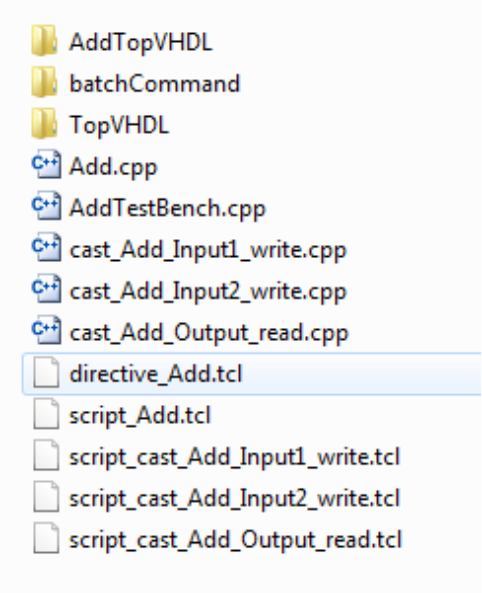


Figure A.7: Step 2: Content of the output folder `HLSBackend`.

2. `Command_Actor.bat` to generate HDL files using Vivado HLS for one actor individually.

- `Actor.cpp`: *C*_HLS code of the corresponding actor –compatible with Vivado HLS
- `Script_Actor.tcl`: will setup the Vivado HLS project: `suProject_Actor`
- `directive_Actor.tcl`: contains Vivado HLS directives for each actor
- `TopVHDL` folder: Contains VHDL files for system-level integration (`NetworkTop.vhd`) and testbench (`Network_TopTestBench.vhd`) for simulating the whole Network
- `ActorTopVHDL` folder: Contains VHDL files for actor-level integration (`ActorTop.vhd`) and testbench (`Actor_TopTestBench.vhd`) for simulating an actor in a standalone fashion
- `ram_tab.vhd`: VHDL model of a dual-port RAM for BRAM inference.

We will explain in what stage each file will be useful in the following steps.

A.4 Compile the *C*-HLS code to VHDL with Vivado HLS

In order to generate hardware components of the whole network `AddOrcc.xdf`, double click on the file `Command.bat` under the `BatchCommand` folder. When running, this batch file will call some files already generated by the Orcc's *C*-HLS Backend. All the VHDL files will be copied under the folder `TopVHDL` as shown in Figure A.8:

A.5 Synthesize the VHDL Using Xilinx ISE

- Create a new folder under the `TopVHDL` folder: name it for example: `AdOrccISE`
- In Xilinx ISE, File >New project (Figure A.9(a))
- Project >Add Source >and select all the VHDL files under the `TopVHDL` folder (Figure A.9(b))
- Set as Top Module the `NetworkTop.vhd` file (Figure A.10(a))
- Run *Synthesize-XST* (Figure A.10(b))
- Implement the design

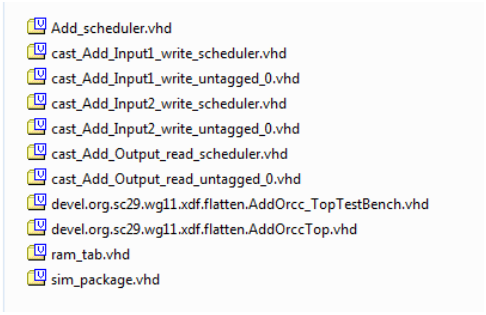
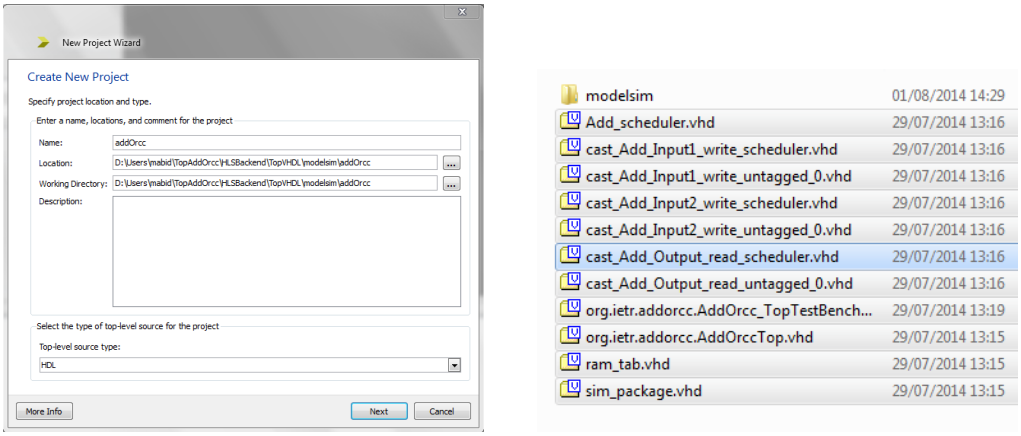
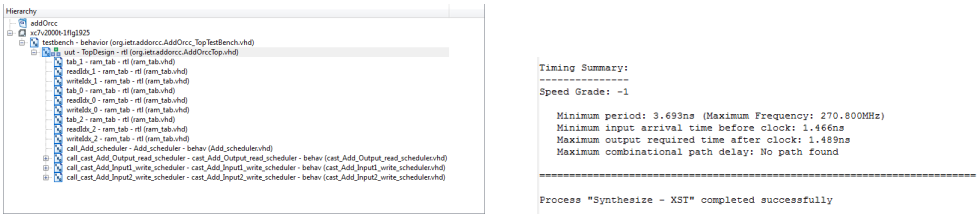


Figure A.8: Step 3: Files resulting from hardware synthesis.



(a) (b)

Figure A.9: Step 4: How to Synthesize the VHDL Using Xilinx ISE.



(a) (b)

Figure A.10: Step 4: How to Synthesize the VHDL Using Xilinx ISE.

B

HEVC Test Sequences

class	Size or type	Sequence name	Frame rate (Fps)	Bit Depth	QP
A	1600p (2K)	Traffic	30	8	22 27 32 37
		PeopleOnStreet	30	8	22 27 32 37
		Nebuta	60	10	22 27 32 37
		StreamLocomotive	60	10	22 27 32 37
B	1080p (HD)	Kimono	24	8	22 27 32 37
		ParkScene	24	8	22 27 32 37
		Cactus	50	8	22 27 32 37
		BQTerrace	60	8	22 27 32 37
		BasketBallDrive	50	8	22 32 37
C	832 × 480 (WVGA)	RaceHorses	30	8	22 27 32 37
		BQMall	60	8	22 27 32 37
		PartyScene	50	8	22 27 32 37
		BasketBallDrill	50	8	22 27 32 37
D	416 × 240 (WQVGA)	RaceHorses	30	8	22 27 32 37

class	Size or type	Sequence name	Frame rate (Fps)	Bit Depth	QP
		BQSquare	60	8	22
					27
					32
					37
		BlowingBubbles	50	8	22
					27
					32
					37
		BasketBallPass	50	8	22
					27
					32
					37
E	720p (Videoconference)	FourPeople	60	8	22
					27
					32
					37
		Jhonny	60	8	22
					27
					32
					37
		KristenAndSara	60	8	22
					27
					32
					37
F	$ScreenCapture$	BasketBallDrillText	50	8	22
					27
					32
					37
		ChinaSpeed	30	8	22
					27
					32
					37
		SlideEditing	30	8	22
					27
					32
					37
		SlideShow	20	8	22
					27
					32
					37

C

Summary of Vivado HLS directives

Directives		Area	Throughput	Latency
Function optimization	Reuse	✓		
	Inline	-	✓	✓
	Instantiate	+	✓	✓
	Dataflow		✓	✓
	Pipeline		✓	
	Latency			
	Interface			
Loop optimization	Unrolling			✓
	Merging			✓
	Flattening			✓
	Dataflow			
	Pipelining		✓	✓
	Dependence			
	Tripcount			
	Latency			
Array optimization	Resource			
	Map	✓		
	Partition		✓	
	Reshape			
	Stream			
Logic structures optimization	Operator selection			
	Controlling hardware resources	✓	✓	✓
	Struct packing	✓	✓	✓
	Expression balancing			✓

D

Résumé en français

D.1 Contexte et Motivation

Cette thèse présente une méthodologie pour la mise en œuvre des algorithmes de compression vidéo sur circuits logiques programmables (Field-Programmable Gate Array (FPGA)). La conception de ces systèmes complexes devient extrêmement difficile en raison de plusieurs facteurs.

Les algorithmes de compression vidéo sont de plus en plus complexes. La compression vidéo est le noyau de la technologie utilisée dans les produits électroniques grand public basés multimédia (c.-à-d. les systèmes multimédia embarqués) tels que les caméras numériques, les systèmes de surveillance vidéo et ainsi de suite. Au fil des ans, les normes Moving Picture Experts Group (MPEG) de codage vidéo ont évolué à partir de MPEG-1, MPEG-4/Advanced Video Coding (AVC) à au codage vidéo haute performance (High-Efficiency Video Coding (HEVC)). En outre, les résolutions vidéo ont augmentées du format Quarter Common Interface Format (QCIF) (144p) à la High Definition (HD) (1080p) à l'Ultra-High Definition (UHD) (4K et 8K), résultant en une augmentation de la compléxité de résolution d'approximativement 1000× par rapport à la QCIF. Outre une résolution plus élevée, la raison principale derrière la complexité croissante des applications de codage vidéo est l'ensemble des outils complexes des codeurs vidéo avancés. Par exemple, à la différence des normes précédentes, le standard de codage vidéo HEVC adopte des techniques de codage très avancées afin d'atteindre un taux de compression élevé pour les résolutions vidéo HD et UHD au prix d'une complexité de calcul additionnelle (approximativement 3× par rapport au H.264).

Le processus de conception des systèmes embarqués est devenu remarquablement difficile. La conception des systèmes embarqués implique la cartographie de l'application cible sur une architecture d'implémentation donnée. Cependant, ces systèmes ont des exigences strictes concernant la taille, la performance, les contraintes temps réel, le délai de mise sur le marché et la consommation d'énergie, etc. Par conséquent, satisfaire ces exigences est une tâche difficile et nécessite des nouvelles méthodologies d'automatisation et des plate-formes de calcul de plus en plus efficaces. Il existe différentes plate-formes matérielles possibles allant des systèmes embarqués à base de processeur (General-Purpose Processor (GPP), Digital Signal Processor (DSP), multiprocessor System On Chip (MPSoC), etc.) aux FPGA et Application-Specific Integrated Circuit (ASIC). Le choix d'un matériel approprié dépend des exigences des applications. Toutefois, afin de traiter les applications de compression vidéo temps réel, qui sont basées sur des algorithmes de calcul intensif, il ya un besoin croissant en puissance de calcul. Alors, quel genre de plate-forme matérielle est le mieux adapté pour les applications en temps réel sous considration? Contrairement aux systèmes embarqués basés processeur, les implémentations matérielles sur FPGA et ASIC se sont révélées être le bon choix en raison de leur architecture massivement parallèle qui résulte en un traitement à grande vitesse.

La conception des systèmes embarqués est confrontée à un écart dans la productivité. Selon la feuille de route pour le progrès technologique (International Technology Roadmap for Semiconductors (ITRS)), le progrés de la productivité de conception ne suit pas le rythme du progrés de la productivité des semi-conducteurs. Cela donne lieu à une augmentation de "l'écart dans la productivité de conception" de façon exponentielle –c.-à-d. la différence entre le taux de croissance des circuits intégrés, mesuré en termes de nombre de portes logiques ou transistors par puce, et le taux de croissance de la productivité du concepteur offerte par les méthodologies et les outils de conception.

Les méthodologies de référence ne sont plus adaptées. Les moyens traditionnels de spécifications des standards MPEG de codage vidéo, basés sur des descriptions textuelles et sur des spécifications monolithiques $C/C++$, ne conviennent plus aux architectures parallèles. D'une part, un tel formalisme de spécification ne permet pas aux concepteurs d'exploiter les points communs clairs entre les différents codecs vidéo, ni au niveau de la spécification, ni au niveau de l'implémentation. D'autre part, la cartographie des spécifications monolithiques $C/C++$ sur des architectures parallèles, tels que les FPGAs, signifie la réécriture du code source complètement afin de distribuer les calculs sur les différentes unités de traitement, ce qui est une tâche fastidieuse et longue. Afin d'améliorer la réutilisation et le délai de mise sur le marché, il ya un grand besoin de développer des méthodologies de conception et de vérification qui permettront d'accélérer le processus de conception et de minimiser l'écart dans la productivité de conception.

D.2 Énoncé du problème et contributions

Beaucoup de questions se posent au sujet des approches appropriées pour combler l'écart dans la productivité de conception ainsi que l'écart entre les spécifications séquentielles traditionnelles et les implémentations parallèles finales. D'une part, selon l' ITRS, l'amélioration de la productivité de conception peut être obtenu en élevant le niveau d'abstraction au-delà du niveau transfert de registres (Register-Transfer Level (RTL)) et en employant des stratégies de conception par réutilisation. D'autre part, la conception au niveau système a émergé comme une nouvelle méthodologie de conception pour combler l'écart entre la spécification et l'implémentation dans les méthodologies traditionnelles. En effet, élever le niveau d'abstraction au niveau système permet au concepteur de gérer la complexité de l'ensemble du système sans tenir compte des détails d'implémentation bas niveau et conduit donc à un nombre réduit de composants à gérer. Cependant, le défi majeur à élever le niveau d'abstraction au niveau système est de traiter avec la complexité d'intégration du système et d' effectuer l'exploration de l' espace de conception (Design-Space Exploration (DSE)), ce qui signifie que *les développeurs ont besoin de savoir comment rassembler les différentes composantes à travers des mécanismes de communication efficaces, tout en permettant des optimisations au niveau système.* En outre, la vérification est essentielle dans le processus de conception au niveau système, ce qui permet d'affirmer que le système répond à ses besoins prévus.
Dans ce contexte, et en connaissant les inconvénients des spécifications monolithiques des standards de codage vidéo, les efforts ont porté sur la standardisation d'une bibliothèque de composants de codage vidéo appelée norme de codage vidéo reconfigurable (Reconfigurable Video Coding (RVC)). Le concept clé derrière la norme est d'être en mesure de concevoir un décodeur à un niveau d'abstraction plus élevé que celui fourni par les spécifications monolithiques actuelles en veillant à l'exploitation du parallélisme, la modularité, la réutilisation et la reconfiguration. Le standard RVC est construit sur la base d'un langage spécifique à un domaine (Domain Specific Language (DSL)) basé flot-de-données connu sous le nom de RVC-CAL, qui est un sous-ensemble de Caltrop Actor Language (CAL). Le standard RVC est basé sur une programmation flot-de-données dynamique. Le modèle de calcul (Model of Computation (MoC)), qui sert à spécifier la manière dont les données sont transférées et traitées, est connu sous le nom de réseau de processus flot-de-données (Dataflow Process Network (DPN)). L'objectif de cette thèse est alors de proposer une nouvelle méthodologie de prototypage rapide sur des FPGAs des programmes flot-de-données basés sur le modèle de calcul DPN. Plusieurs questions pourraient être soulevées à savoir comment traduire les programmes fondés sur le modèle de calcul DPN en descriptions RTL appropriées pour une implémentation matérielle efficace, tout en réduisant la complexité

et le délai de mise sur le marché, et en obtenant des implémentations avec des performances efficaces. Plusieurs travaux ont cherché à répondre à ces questions, mais ont fourni seulement des solutions partielles pour la synthèse au niveau système.
Motivés par ces développements, nos contributions face aux défis de l'implémentation des programmes flot-de-données dynamiques sur FPGA sont comme suit.

- Premièrement, nous proposons un nouveau flot de conception automatisé pour le prototypage rapide des décodeurs vidéo basés sur RVC, au moyen duquel un modèle spécifié dans le langage flot-de-données RVC-CAL au niveau système est rapidement traduit en une implémentation matérielle. En effet, nous concevons les algorithmes de compression vidéo en utilisant le langage orienté acteur selon la norme RVC. Une fois la conception réalisée, nous utilisons une infrastructure de compilation flot-de-données appelée Open RVC-CAL Compiler (Orcc) pour générer un code basé sur le language *C*. Par la suite, un outil de Xilinx appelé Vivado High-Level Synthesis (HLS) est utilisé pour une génération automatique d'une implémentation matérielle synthétisable.

- Ensuite, nous proposons une nouvelle méthode de synthèse de l'interface qui permet l'amélioration de la mise en œuvre des voies de communication entre les composants et en conséquence l'amélioration des politiques d'ordonnacement, visant ainsi à optimiser les mesures de performance tels que la latence et le débit des décodeurs vidéo basés flot-de-données. Par consquent, une nouvelle implémentation au niveau système est élaborée sur la base de cette implémentation optimisée de la communication et des mécanismes d'ordonnancement.

- Ensuite, nous étudions les techniques d'aide à l'exploration de l'espace de conception (DSE) afin d'atteindre des implémentations de haute performance en exploitant le parallélisme au niveau tâche ainsi que le parallélisme au niveau données, et ceci au niveau système.

- Enfin, nous présentons un cadre pour la vérification au niveau système ou au niveau composant. Par consquent, nous démontrons l'efficacité de notre méthode de prototypage rapide en l'appliquant à une implémentation RVC-CAL du décodeur HEVC, qui s'est avéré une tâche très difficile, car le décodeur HEVC implique généralement une grande complexité de calcul et des quantités massives de traitement de données.

D.3 Organisation du rapport de thèse

Cette thèse est structurée comme suit.
La première partie décrit le contexte de l'étude, y compris son cadre théorique. Le chapitre 2 comporte un aperçu des tendances et des défis rencontrés lors de la conception des systèmes embarqués et de l'émergence de la conception au niveau système des systèmes embarqués. Le chapitre 3 étudie les propriétés de base de la programmation flot-de-données, introduit le cadre MPEG-RVC ainsi que son langage de programmation de référence et la sémantique du modèle de calcul DPN. Ensuite, il résume les différentes approches existantes pour la génération du code HDL à partir des représentations flot-de-données.
La deuxième partie présente les principales contributions de cette thèse. Dans le chapitre 4, une méthodologie de prototypage rapide pour les programmes basés sur DPN est présentée. Le flot de conception proposé combine un compilateur flot-de-données pour générer des descriptions de synthèse de haut niveau (HLS) à base du code *C* à partir d'une description flot-de-données et un synthesizeur *C*-à-RTL pour générer des descriptions RTL. Les résultats obtenus sur une description RVC-CAL du décodeur HEVC sont discutées. Le chapitre 5 présente quelques techniques d'optimisation en proposant tout d'abord une nouvelle méthode de la synthèse de l'interface et ensuite, en exploitant toutes les fonctionnalités de la programmation flot-de-données dynamique. Le chapitre 6 conclut les deux parties de cette thèse et discute les perspectives de travaux futurs.
La troisième partie fournit des informations supplémentaires à cette thèse. Dans l'annexe A, nous présentons un guide pour la génération de code en langage de description de matériel (Hardware Description Language (HDL)) à partir de programmes flot-de-données selon le flot de conception proposé. Les annexes **??** présentent toutes les séquences d'essais disponibles du décodeur HEVC et un résumé des différentes directives de Vivado HLS, respectivement.

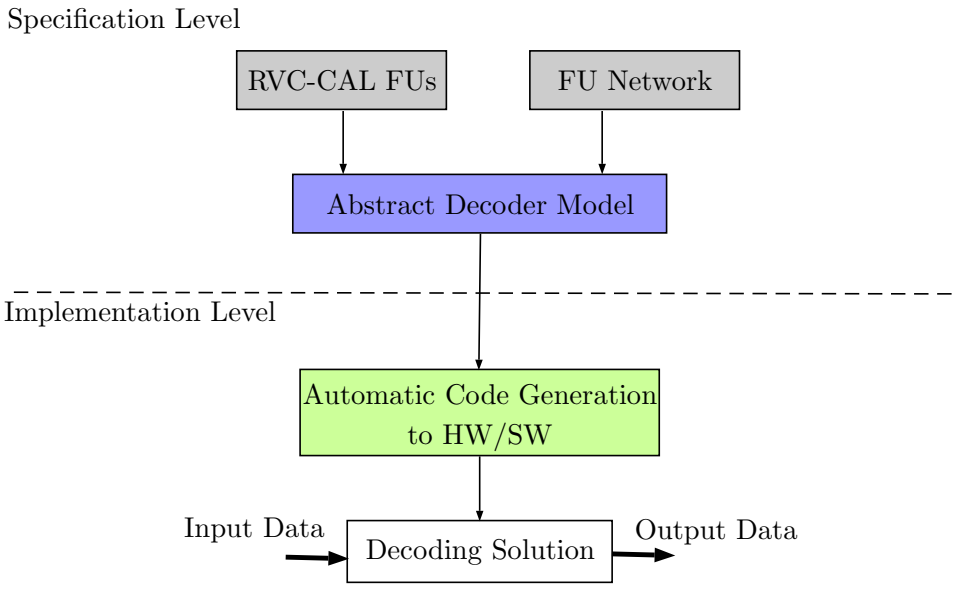


Figure D.1: La norme RVC: La partie supérieure est le processus standard d'élaboration d'une spécification abstraite, la partie inférieure est le processus non-standard de génération des implémentations multi-cibles à partir de la spécification standard.

D.4 État de l'art

Dans cette section, nous donnons un bref aperçu des principaux concepts de base pour comprendre le travail présenté dans cette thèse.

D.4.1 Le paradigme de programmation flot-de-données

Contrairement au paradigme de programmation séquentielle, l'approche flot-de-données est un paradigme de programmation qui modélise un programme comme un graphe orienté dans lequel les nœuds correspondent à des unités de calcul et les arêtes représentent la direction des données circulant entre les nœuds. D'une part, le comportement fonctionnel de chaque unité de calcul est autonome et indépendant des autres unités de calcul, assurant ainsi la modularité, la réutilisation et la reconfiguration et facilitant l'exploitation du parallélisme. D'autre part, la sémantique de communication et de traitement des unités fonctionnelles (Functional Unit (FU)) est définie par un modèle de calcul (Model of Computation (MoC)) tels que Kahn Process Network (KPN), Synchronous Dataflow (SDF) et Dataflow Process Network (DPN) [Lee and Parks, 1995b]. Nous examinons brièvement le modèle de calcul DPN car il constitue le modèle de calcul de base utilisé par CAL, qui est un super-ensemble du langage RVC-CAL normalisé dans le cadre RVC.

D.4.2 La norme MPEG-RVC

Dans cette section, nous donnons un bref aperçu des concepts et les outils mis en place dans le cadre RVC. Tout au long de cette étude, nous mettons en évidence les avantages obtenus de l'adoption de RVC du point de vue implémentation matérielle. Basé sur le paradigme flot-de-données, le comité de normalisation MPEG standardise la norme RVC en 2009 pour la spécification des codecs vidéo. Le but de RVC est de remédier aux limitations des spécifications monolithiques (généralement sous la forme de programmes $C/C++$) qui ne peuvent plus faire face à la complexité croissante des normes de codage vidéo ni exprimer le parallélisme intrinsèque de ces applications. En outre, ces spécifications monolithiques ne permettent pas aux concepteurs d'exploiter les points communs entre les différents codecs et de produire des spécifications en temps opportun.
Essentiellement, modularité, réutilisabilité et reconfigurabilité sont les caractéristiques de la norme RVC. La figure D.1 illustre comment un décodeur vidéo est conçu à un haut niveau d'abstraction et comment les implémentations cibles sont génrés dans le cadre RVC. Au niveau

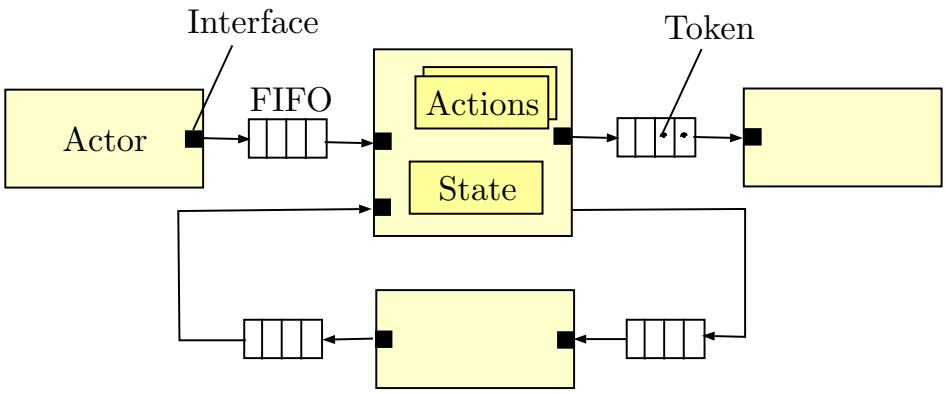


Figure D.2: Un modèle DPN est conçu comme un graphe orienté composé de sommets (c.-à-d. acteurs) et les bords représentent des canaux de communication unidirectionnels basés sur le principe FIFO.

spécification (la partie normative), le comportement fonctionnel et les entrées/sorties des unités fonctionnelles (FUs) sont d'abord décritent en utilisant un langage de programmation orienté acteur appelé RVC-CAL défini dans MPEG-B pt.4 (ISO/IEC 23001-4). La norme RVC fournit également une librairie normative d'outils vidéo (Video Tool Library (VTL)) définie dans MPEG-C pt.4 (ISO/IEC 23002-4), qui contient toutes les unités fonctionnelles (FUs) nécessaires pour décrire toutes les normes MPEG de codage vidéo. Les connexions entre les FUs sont ensuite décritent pour former un réseau de FUs, exprimé en langage (FU Network Language (FNL)), qui constitue la configuration d'un décodeur vidéo. Le langage FNL permet également une configuration hiérarchique, à savoir un FU peut être décrit comme une composition d'autres FUs. Enfin, les FUs et le réseau de FUs sont instanciés pour former un modèle de décodage abstrait (Abstract Decoder Model (ADM)), qui est un modèle de comportement normatif du décodeur. Au niveau de l'implémentation, l'ADM est utilisé pour créer automatiquement des implémentations pour de multiples plates-formes cibles (logicielles et matérielles). Dans ce cadre, plusieurs outils de synthèse permettant la génération automatique d'une spécification RVC existent comme un support non-normatif de la norme RVC. Dans ce qui suit, nous nous concentrons sur l'implémentation matérielle dans le cadre RVC.

D.4.3 Le langage de programmation flot-de-données RVC-CAL et son modèle de calcul

Le standard RVC est construit sur la base d'un langage spécifique à un domaine (DSL) basé flot-de-données connu sous le nom RVC-CAL, un sous-ensemble de CAL [Eker and Janneck, 2003]. Le modèle de calcul (MoC) utilisé qui précise la façon dont les données sont transférées et traitées est connu sous le nom de DPN [Lee and Parks, 1995a], un cas particulier de KPN [Kahn, 1974]. Dans ce modèle, les FUs sont implementées en tant qu'acteurs contenant un certain nombre d'actions et d'états internes. Dans DPN, le comportement des acteurs est dépendant des données et les états internes d'un acteur sont complètement encapsulés et ne peuvent être partagés avec d'autres acteurs. Ainsi, les acteurs s'exécutent simultanment et communiquent avec les autres exclusivement à travers les ports (interfaces), via le passage de données le long des canaux de communication illimités, unidirectionnels et basés sur le principe FIFO comme illustré sur la figure D.2.
Cependant, les actions dans un acteur sont atomiques, ce qui signifie qu'une fois qu'une action s'exécute, aucune autre action ne peut s'exécuter jusqu'à ce que la précédente ait terminée. Les données qui sont échangées entre les acteurs sont appelés jetons. L'exécution d'une action est un quantum indivisible de calcul qui correspond à une fonction de mappage des jetons d'entrée vers des jetons de sortie. Cette cartographie est composée de trois étapes ordonnées et indivisibles:

- lire un nombre de jetons à partir des ports d'entrée de l'acteur;
- exécuter des transformations sur l'état interne de l'acteur (à savoir la procédure de calcul);
- écrire un nombre de jetons sur les ports de sortie de l'acteur.

Chaque action d'un acteur peut s'exécuter selon quatre différentes conditions, appelées règles de tir:

1. disponibilité des jetons d'entrée (c.-à-d. s'il y a suffisamment de jetons sur tous les ports d'entrée);

2. les conditions d'exécution d'une action "*les guards*" (qui évaluent l'état interne d'un acteur et jettent un coup d'oeil dans la valeur des jetons d'entrée);

3. la machine d'états finis (Finite-State Machine (FSM)) qui ordonnance les actions d'un acteur;

4. les priorités (qui décrivent l'action qui doit être congédiée lorsque plusieurs actions sont éligibles à s'exècuter).

D.4.4 Travaux connexes en génération de code HDL à partir de programmes RVC-CAL

La complexité de la conception et les processus longs de vérification créent un goulot d'étranglement pour les applications de codage vidéo. Afin de diminuer le délai de mise sur le marché, de nombreuses solutions ont été mises au point en élevant le niveau d'abstraction au niveau système électronique (Electronic System Level (ESL)) [Martin et al., 2007].

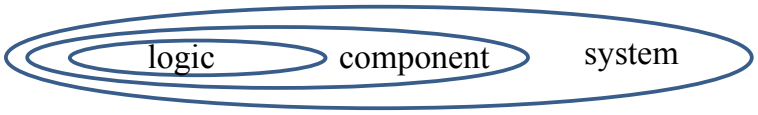


Figure D.3: Les différents niveaux d'abstraction.

Les niveaux d'abstraction communs qui sont utilisés pour la conception des circuits intégrés numériques sont illustrés sur la figure D.3. Le niveau d'abstraction le plus élevé est le niveau système, auquel la conception est faite en prenant l'ensemble du système en considération, non seulement des composants individuels. En d'autres termes, les interactions entre les différents composants sont examinés à un niveau d'abstraction plus élevé . Ensuite, au niveau composant, une description algorithmique dans un langage haut niveau est synthétisée vers une description RTL. Cette étape est communément appelée synthèse haut niveau (HLS). Dans ce qui suit, nous examinons les différents outils disponibles pour la génération de code matériel, soit au niveau composant ou au niveau système.

D.4.4.1 Conception au niveau composant

Au niveau composant, il existe plusieurs outils pour effectuer automatiquement la synthèse haut niveau (HLS). En général, *C* est le langage de haut niveau utilisé. Ici, la HLS prend comme entrée un modèle décrit en *C*, *C++*, ou SystemC, et en tant que sortie, elle génère une représentation RTL correspondante en langage de description de matériel (HDL) tels que VHDL ou Verilog. Dans ce cas, nous discutons des outils HLS comme Catapult C, C2H, Synphony, GAUT, etc. Cependant, l'objectif de la compilation des applications réelles, décritent dans un langage tel que *C*, en implémentations matérielles efficaces, s'accompagne de très fortes limitations puisque l'ensemble du système n'est pas pris en considération.

D.4.4.2 Conception au niveau système

Nous présentons les flots de conception qui existent pour la mis en œuvre des applications RVC sur des plates-formes matérielles en utilisant deux outils front-end: OpenDF [Bhattacharyya et al., 2008] et Orcc [Wipliez, 2010].

OpenDF OpenDF agit comme un outil front-end et génère un code XML Language-Independent Model (XLIM) à partir d'un modèle CAL. Ensuite OpenForge agit comme un outil back-end pour générer un code HDL à partir du modèle XLIM. Le flot de conception de CAL vers HDL dans OpenDF est également connu sous le nom CAL2HDL [Janneck et al., 2008]. Le principal problème avec OpenDF est que la génération de code ne fonctionne pas avec toutes les structures RVC-CAL et le code généré est si difficile à gérer et corriger. Dans le but de surmonter ces problèmes, OpenDF a été remplacé par le compilateur Orcc.

Orcc est un environnement de développement intégré libre basé sur Eclipse et dédié à la programmation flot-de-données. Le but principal de Orcc est de fournir aux développeurs une infrastructure de compilation pour permettre la génération de code logiciel/matériel à partir de descriptions flot-de-données. Dans ce cadre, l'approche proposée par Siret et al. [Siret et al., 2012] offre un nouveau générateur de code matériel en ajoutant un nouveau backend au compilateur Orcc. Malheureusement, le travail n'a pas été finalisé. Une autre approche [Bezati et al., 2011] cherche à utiliser l'outil OpenForge comme backend du modèle XLIM généré par Orcc. Le flot de conception de CAL vers HDL dans Orcc est également connu sous le nom ORC2HDL. La limitation de cette méthodologie est le manque de support aux actions multi-jetons dans les programmes RVC-CAL. Bien que la solution proposée par Jerbi et al. [Jerbi et al., 2012] pour surmonter cette limitation, qui est une transformation automatique de programmes RVC-CAL multicadence à des programmes à taux unique, elle conduit à un code résultant complexe et la réduction de la performance. Des travaux récents [Bezati et al., 2013] ont cherché à améliorer le flot de conception de ORC2HDL, en alimentant OpenForge par une représentation intermédiaire (Intermediate Representation (IR)) générée par Orcc, connu sous le nom xronos. Le principal problème avec cette approche est la nécessité de changer certaines constructions dans le code RVC-CAL initial afin de pouvoir faire la synthèse.
Orcc est actuellement le plus largement utilisé pour la génération de code dans la communauté RVC et il est également le choix de notre travail rapporté dans cette thèse.

D.5 Présentation du flot de conception niveau système proposé

Nous avons proposé dans notre article de conférence [Abid et al., 2013] un flot de conception complet illustré dans la figure D.4. Le compilateur Orcc génère un modèle basé sur *C* à partir de programmes RVC-CAL, que nous avons qualifié backend *C*-HLS. Ensuite, Vivado HLS [1] de Xilinx est utilisé comme outil HLS qui compile automatiquement le code basé sur *C* dans une description RTL en HDL. Il s'avère que Vivado HLS a un avantage significatif puisqu'il accélère la productivité pour les FPGAs Xilinx série 7 et pour de nombreuses générations des FPGAs à venir. Le premier défi clé dans la mise en œuvre du flot de conception proposé est la génération automatique d'un code *C* conforme à Vivado HLS et qui respecte la sémantique du modèle de calcul DPN. Le deuxième défi clé est la synthèse automatique du système y compris la synthèse de la communication. Tout au long du chapitre 4, nous avons détaillé les spécifications de la génération du code *C*-HLS tout en respectant la sémantique du modèle de calcul DPN, afin de garder le même comportement de l'acteur dans la description matérielle. Nous avons fourni la méthodologie utilisée pour établir le niveau système en connectant avec précision les différents composants matériels générés par Vivado HLS avec les composants FIFOs correspondants (figure D.5).
La première caractéristique du backend *C*-HLS de Orcc est le fait qu'il ne contient pas des constructions qui ne sont pas synthétisables telles que l'allocation dynamique de mémoire et les pointeurs puisque les descriptions RVC-CAL ne supportent pas ce type de constructions. En outre, le backend *C*-HLS respecte la sémantique du modèle de calcul DPN à travers la gestion de la FIFO expliquée en détails dans ce qui suit.
En effet, l'accés aux FIFOs sont accomplies au moyen des méthodes definies dans la classe *hls::stream* de Vivado HLS, utilisée pour définir les interfaces (**streaming explicite**). Cependant, la FIFO renvoie seulement deux informations sur son état: plein ou vide. L'information sur le nombre de jetons présents dans la FIFO d'entrée et leurs valeurs n'est pas disponible. Afin

[1] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/

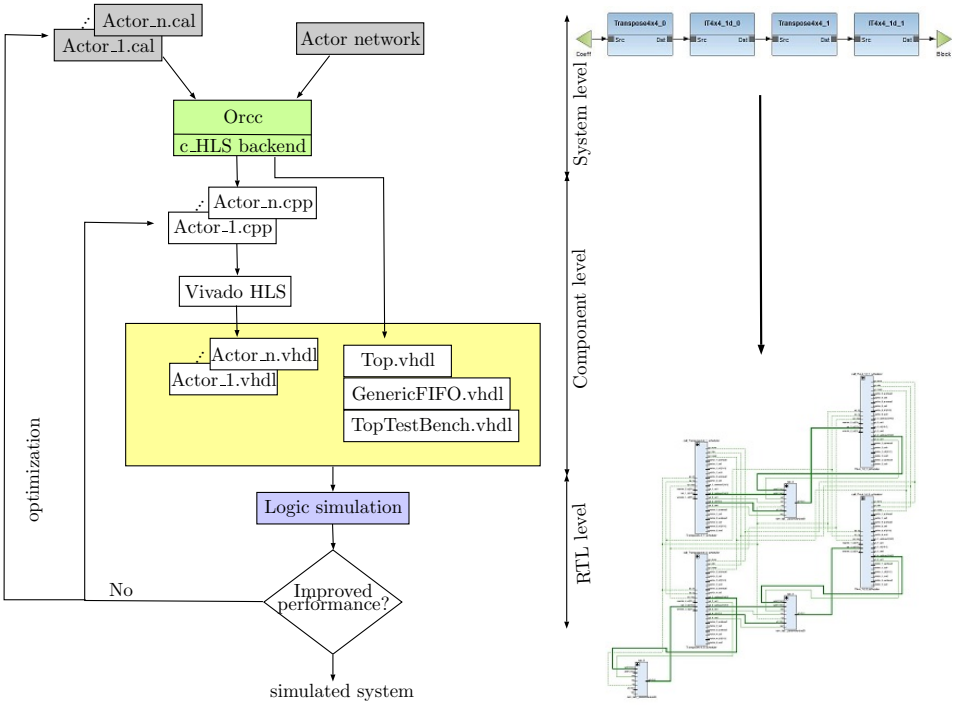


Figure D.4: Flot de conception niveau système proposé.

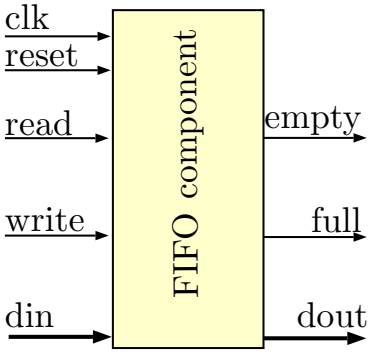


Figure D.5: Implémentation matérielle de la FIFO.

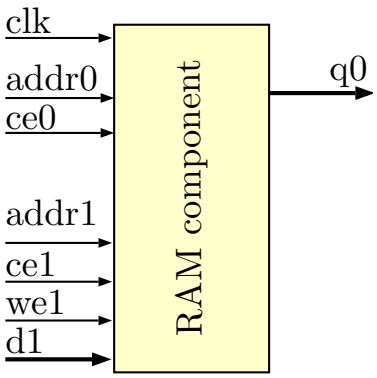


Figure D.6: Implémentation matérielle de la RAM.

de respecter la sémantique du modèle de calcul DPN, la solution [Jerbi et al., 2012] était de créer des buffers circulaires internes pour chaque port d'entrée où les jetons pourront être stockés, et qui sont gérés par des indexes de lecture et d'écriture. Une action est crée juste pour lire les données de la FIFO et les stocker dans le buffer circulaire interne en incrémentant les indexes de lecture. Plus tard, la consommation des données à partir des buffers incrémente les indexes d'écriture. Par conséquent, la différence entre les indexes de lecture et d'écriture correspond au nombre de jetons disponibles dans chaque buffer et toutes les règles de tir des actions sont liées à cette différence. La taille de ces buffers internes est la pluissance entière de 2 la plus proche du nombre de jetons de lecture tandis que les FIFOs créés sont par défaut implémentées avec une profondeur de 1.

D.6 Problématique du flot de conception proposé

Bien que cette implémentation respecte la sémantique du modèle de calcul DPN, le fait d'ajouter des buffers internes pour chaque port d'entrée, alourdissent la structure de l'acteur. En effet, pour chaque port d'entrée, on ajoute une action pour consommer les jetons d'une FIFO d'entrée et les stocker dans une mémoire tampon locale. Ensuite, on ajoute une action qui effectue le calcul de base et remplit une mémoire tampon locale pour chaque port de sortie. Enfin, on ajoute une action qui écrit les jetons de la mémoire tampon locale vers les FIFOs de sortie, ainsi qu'une machine d'état pour l'ordonnancement de ces actions, ce qui engendre des copies supplémentaires entre les FIFOs et les buffers internes. En outre, la consommation et la production de jetons n'est pas effectuée en parallèle en raison du modèle d'exécution séquentiel des actions d'un acteur, ce qui augmente la latence et l'utilisation des ressources et réduit le débit.

D.7 Optimization de l'interface de communication

Cette section traite le goulot d'étranglement des performances introduit par l'infrastructure de communication proposée. Nous proposons une implémentation optimisée de l'infrastructure de communication (**streaming explicite**) dans la conception proposée précédemment, afin de minimiser les frais généraux engendrés par l'infrastructure de communication et d'ordonnancement. Ainsi, nous avons amélioré le backend *C*-HLS en utilisant le **streaming implicite** plutôt que explicite. Les ports de l'interface sont déclarés comme des buffers circulaires unidimensionnels externes permettant ainsi l'accès à la mémoire externe. Au niveau RTL, ces buffers circulaires sont convertis en blocs de mémoire (Random-Access Memory (RAM)) par la synthèse haut niveau (HLS). Ce type d'interface est utilisé pour communiquer avec des éléments de mémoire RAM comme ilustré à la figure D.6.
L'accés aux FIFOs est effectué en accédant directement au contenu des buffers puisqu' ils sont mis en œuvre comme mémoire partagée. En outre, l'utilisant des buffers circulaires pour implémenter les FIFOs exige une gestion efficace des indexes. Ainsi, chaque acteur qui écrit/lit dans/du buffer circulaire a son propre indexe d'écriture/lecture local. Afin d'éviter les situations de compétition, les indexes sont incrémentés une seule fois à la fin de l'action. Dans le but de rendre l'état des

Table D.1: Résultats temporels de l'implémentation du décodeur RVC CAL HEVC selon deux flots de conception: *Stream design* vis-à-vis *RAM design* pour une séquence vidéo de 5 images et une taille de FIFO de 16384.

	Stream design	RAM design
Latency (ms)	248, 10	64, 83
Sample Rate (Msps)	0, 54	2, 71
Throughput (Fps)	3, 66	18, 11

indexes visible par d'autres acteurs, les indexes locaux sont transmis à la fin de l'action vers des interfaces externes déclarées comme des tableaux unidimensionnels de taille 1. Ainsi, chaque acteur gère son propre indexe pour la lecture et/ou écriture, et peut avoir accès aux indexes des autres acteurs.
L'ordonnanceur des actions est implémenté par une fonction, qui évalue les règles de tir, à savoir les valeurs de jetons qui devrait être disponibles dans les canaux d'entrée ainsi que leurs nombres. Cependant, l'information sur le nombre de jetons disponibles dans les canaux d'entrée n'est pas disponible lorsqu'on traite avec des tampons. La comparaison des indexes de lecture et d'écriture associées à chaque tampon est suffisante pour reconnaître l'état de la FIFO. Par conséquent, des tests, pour détecter le moment auquel le nombre de jetions entrants nécessaires est disponible ou quand une FIFO de sortie est pleine, sont ajoutés dans l'ordonnanceur d'actions. Ces tests sont réalisés au moyen de la différence entre les indexes de lecture et d'écriture.

D.8 Cas d'étude: le décodeur HEVC

Pour démontrer l'applicabilité de notre méthodologie de prototypage rapide proposée, nous implémentons automatiquement une description RVC-CAL du décodeur HEVC en FPGA selon le flot de conception niveau système proposé de la figure D.4. Nous avons choisi la norme HEVC vu que c'est la dernière norme au sein du groupe Joint Collaborative Team on Video Coding (JCT–VC). Le décodeur HEVC implique l'utilisation d'algorithmes complexes consommant un flux de bits en entrée, et produisant des données vidéo en sortie (Figure D.7). Une représentation graphique de la description RVC-CAL du décodeur HEVC est illustrée à la figure D.8, qui totalise 32 acteurs. Chaque acteur est mappé à un bloc fonctionnel du décodeur commun. De cette manière, l'*Algo_Parser* correspond au décodeur entropique, qui extrait les valeurs nécessaires pour le traitement du prochain flux de données compressées. L'*XiT*, qui est à son tour hiérarchique, met en œuvre la transformée inverse et la quantification. L'*IntraPrediction* correspond à la prédiction spatiale. L'*InterPrediction* correspond à la prédiction temporelle. Le *SelectCu* calcule la reconstruction de l'image. Le *GenerateInfo* obtient principalement les vecteurs de mouvement. Le *DecodingPictureBuffer* correspond au tampon d'images décodées. Deux filtres additionnels sont aussi définis, le *Deblocking Filter (DBF)* et le *Sampling Adaptative offset (SAO)*. Le premier a pour objectif de réduire la complexité. Le second filtre est appliqué après le deblocking et ajoute un offset en fonction de la valeur du pixel et des caractéristiques de la région de l'image.
Pour les résultats expérimentaux, nous avons sélectionné 4 séquences vidéo à partir de la base de données standard de HEVC. Pour l'implémentation FPGA, nous avons ciblé une plate-forme Xilinx Virtex–7 (XC7V2000T package FLG1925–1) en utilisant Vivado HLS (Version 2014.3).
Afin de quantifier les performances de notre flot de conception proposé, trois indicateurs de performance sont considérés qui sont le débit, la latence et la surface.
Pour démontrer l'amélioration de la performance, nous avons implémenté la description RVC CAL du décodeur HEVC selon le flot de conception **streaming explicite** qu'on le nomme *Stream design* d'une part et le flot de conception **streaming implicite** qu'on le nomme *RAM design* d'autre part. Le débit et la latence de ces 2 designs sont compars dans le tableau D.1. Les résultats de la simulation montrent que l'optimisation proposée avec le *RAM design* a augmenté le débit d'un facteur d'accélération de $5,2\times$ et réduit la latence d'un facteur d'accélération de $3,8\times$ par rapport à l'implémentation du *Stream design*. En effet, les résultats dépendent largement du remplacement de la copie des données entre les FIFOs et les tampons internes dans le *Stream design*, par une mémoire partagée dans la conception optimisée *RAM design*.

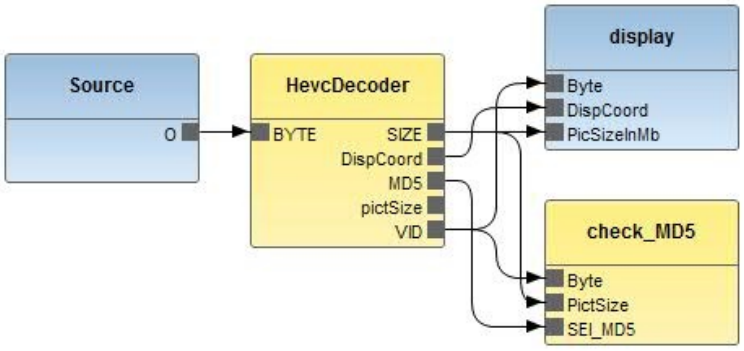


Figure D.7: Description RVC CAL au plus haut niveau du decodeur HEVC.

D.9 Conclusion et perspectives

Dans cette thèse, nous avons proposé une implémentation matérielle optimisée des canaux de communication dans les programmes RVC-CAL en utilisant une mémoire partagée (RAM) qui se comporte comme un buffer circulaire. Les résultats de simulation du décodeur HEVC avec le flot de conception optimisé ont montré que l'implémentation proposée avec les Blocs RAM a augmenté le débit et réduit la latence par rapport au flot de conception état-de-l'art. En outre, le flot de conception proposé est entièrement automatique quel que soit la complexité de l'application, ce qui conduit à un gain en temps de développement par rapport à l'approche manuelle. Par ailleurs, une infrastructure de test complète a été mise en œuvre dans Orcc, qui permet de simuler et analyser le design à un niveau de granularité souhaité (système, composant ou action). Un autre élément clé de notre flot de conception proposé est la capacité d'améliorer les performances par refactoring des programmes RVC-CAL. En d'autres termes, la modélisation des applications de codage vidéo au niveau système dans le cadre RVC facilite l'optimisation au niveau système et composant en faveur de l'implémentation matérielle en faisant abstraction aux détails bas niveau.
Les directions pour des travaux futurs incluent l'application d'optimisations basées sur les directives Vivado HLS pour parvenir à une implémentation matérielle avec un débit plus élevé et une latence plus faible. En outre, l'implémentation d'une mémoire partagée est le point de départ qui permet une conception conjointe (co-design matériel/logiciel) pour les plateformes à base de FPGA.

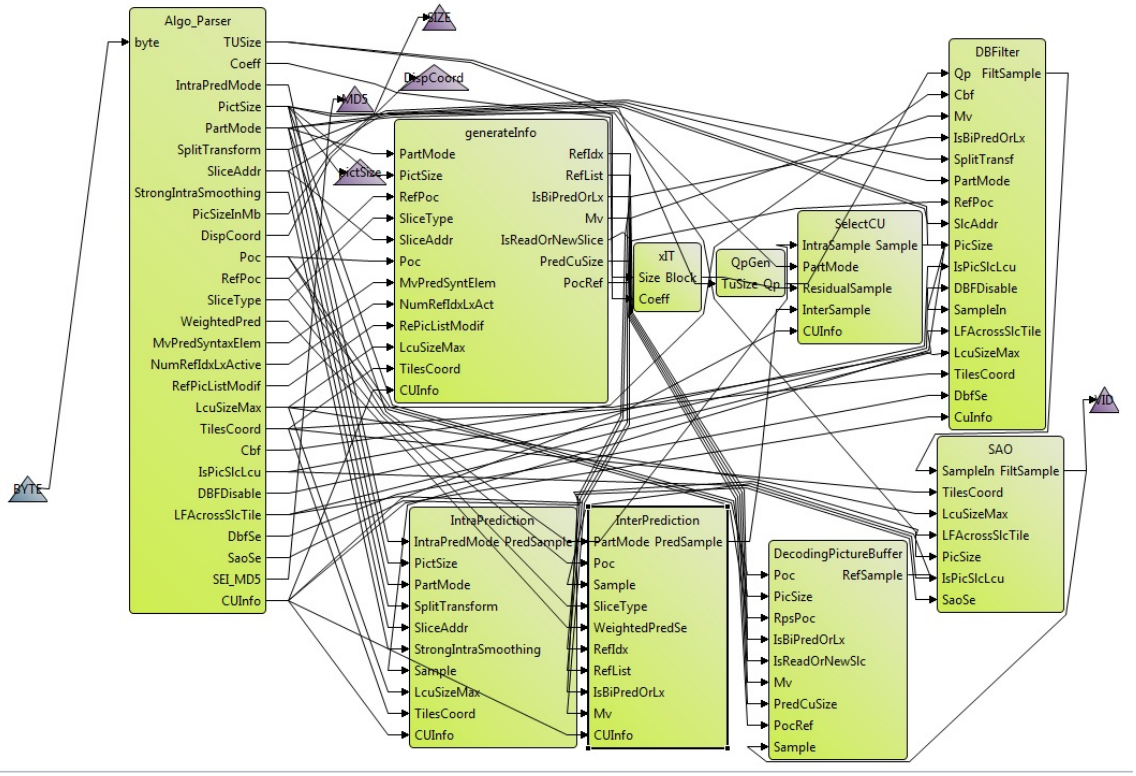


Figure D.8: The RVC FNL description of the HEVC Decoder FU.

Bibliography

[viv] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/. 127

[Abid et al., 2013] M. Abid, K. Jerbi, M. Raulet, O. Deforges, and M. Abid. System level synthesis of dataflow programs: HEVC decoder case study. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pages 1–6, May 2013. 121, 127

[Bezati et al., 2011] 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. doi: 10.1109/DASIP.2011.6136877. 121, 127

[Bezati et al., 2013] E. Bezati, M. Mattavelli, and J. Janneck. High-Level Synthesis of Dataflow Programs for Signal Processing Systems. In *8th International Symposium on Image and Signal Processing and Analysis (ISPA 2013)*, 2013. 121, 127

[Bhattacharyya et al., 2008] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet. OpenDF – A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems. In *First Swedish Workshop on Multi-Core Computing*, page CD, Ronneby, Suède, 2008. 120, 127

[Eker and Janneck, 2003] J. Eker and J. W. Janneck. CAL Language Report Specification of the CAL Actor Language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003. 119, 127

[Janneck et al., 2008] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 287 – 292, Washington, États-Unis, 2008. doi: 10.1109/SIPS.2008.4671777. 121, 127

[Jerbi et al., 2012] K. Jerbi, M. Raulet, O. Deforges, and M. Abid. Automatic generation of synthesizable hardware implementation from high level RVC-cal description. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1597–1600, March 2012. doi: 10.1109/ICASSP.2012.6288199. 121, 123, 127

[Kahn, 1974] G. Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974. 119, 127

[Lee and Parks, 1995a] E. A. Lee and T. Parks. Dataflow Process Networks. In *Proceedings of the IEEE*, pages 773–799, 1995a. 119, 127

[Lee and Parks, 1995b] E. A. Lee and T. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995b. 118, 127

[Martin et al., 2007] G. Martin, B. Bailey, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2007. ISBN 9780080488837. 120, 127

[Siret et al., 2012] N. Siret, M. Wipliez, J. F. Nezan, and F. Palumbo. Generation of Efficient High-Level Hardware Code from Dataflow Programs. In *Proceedings of Design, Automation and test in Europe (DATE)*, 2012. 121, 127

[Wipliez, 2010] M. Wipliez. *Infrastructure de compilation pour des programmes flux de données*. PhD thesis, INSA de Rennes, Dec. 2010. 120, 127

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Synthèse matérielle au niveau système des programmes flots-de-données: étude de cas du décodeur HEVC

Nom Prénom de l'auteur : ABID MARIEM

Membres du jury :
- Madame QUEUDET Audrey
- Monsieur ABID Mohamed
- Monsieur DEFORGES Olivier
- Monsieur AKIL Mohamed
- Monsieur AMMARI Ahmed Chiheb
- Monsieur ATRI Mohamed

Président du jury : M. AKIL

Date de la soutenance : 28 Avril 2016

Reproduction de la these soutenue

Thèse pouvant être reproduite en l'état
Thèse pouvant être reproduite après corrections suggérées

Fait à Rennes, le 28 Avril 2016

Signature du président de jury

Le Directeur,

M'hamed DRISSI

INSTITUT NATIONAL des SCIENCES APPLIQUÉES · RENNES

Résumé

Les applications de traitement d'image et vidéo sont caractérisées par le traitement d'une grande quantité de données. La conception de ces applications complexes, avec des méthodologies de conception traditionnelles bas niveau, provoque l'augmentation des coûts de développement.

Afin de résoudre ces défis, des outils de synthèse haut niveau ont été proposés. Le principe de base est de modéliser le comportement de l'ensemble du système en utilisant des spécifications haut niveau afin de permettre la synthèse automatique vers des spécifications bas niveau pour implémentation efficace en FPGA.

Cependant, l'inconvénient principal de ces outils de synthèse haut niveau est le manque de prise en compte de la totalité du système, c.-à-d. la création de la communication entre les différents composants pour atteindre le niveau système n'est pas considérée.

Le but de cette thèse est d'élever le niveau d'abstraction dans la conception des systèmes embarqués au niveau système. Nous proposons un flot de conception qui permet une synthèse matérielle efficace des applications de traitement vidéo décrites en utilisant un langage spécifique à un domaine pour la programmation flot-de-données. Le flot de conception combine un compilateur flot-de-données pour générer des descriptions à base de code C et un synthétiseur pour générer des descriptions niveau de transfert de registre.

Le défi majeur de l'implémentation en FPGA des canaux de communication des programmes flot-de-données basés sur un modèle de calcul est la minimisation des frais généraux de la communication. Pour cela, nous avons introduit une nouvelle approche de synthèse de l'interface qui mappe les grandes quantités des données vidéo, à travers des mémoires partagées sur FPGA. Ce qui conduit à une diminution considérable de la latence et une augmentation du débit. Ces résultats ont été démontrés sur la synthèse matérielle du standard vidéo émergent High-Efficiency Video Coding (HEVC).

Abstract

Image and video processing applications are characterized by the processing of a huge amount of data. The design of such complex applications with traditional design methodologies, which are at low-level of abstraction, causes increasing development costs.

In order to resolve the above mentioned challenges, Electronic System Level (ESL) synthesis or High-Level Synthesis (HLS) tools were proposed. The basic premise is to model the behavior of the entire system using high-level specifications, and to enable the automatic synthesis to low-level specifications for efficient implementation in Field-Programmable Gate Array (FPGA).

However, the main downside of the HLS tools is the lack of the entire system consideration, i.e. the establishment of the communications between these components to achieve the system-level is not yet considered.

The purpose of this thesis is to raise the level of abstraction in the design of embedded systems to the system-level. A novel design flow was proposed that enables an efficient hardware implementation of video processing applications described using a Domain Specific Language (DSL) for dataflow programming. The design flow combines a dataflow compiler for generating C-based HLS descriptions from a dataflow description and a C-to-gate synthesizer for generating Register Transfer Level (RTL) descriptions.

The challenge of implementing the communication channels of dataflow programs relying on Model of Computation (MoC) in FPGA is the minimization of the communication overhead. In this issue, we introduced a new interface synthesis approach that maps the large amounts of data that multimedia and image processing applications process, to shared memories on the FPGA. This leads to a tremendous decrease in the latency and an increase in the throughput. These results were demonstrated upon the hardware synthesis of the emerging High-Efficiency Video Coding (HEVC) standard.

INSA RENNES

N° d'ordre : 16ISAR 07 / D16 - 07

Institut National des Sciences Appliquées de Rennes
20, Avenue des Buttes de Coëmes • CS 70839 • F-35708 Rennes Cedex 7
Tel : 02 23 23 82 00 - Fax : 02 23 23 83 96