Dissertation

An Enhanced Hardware Description Language Implementation for Improved Design-Space Exploration in High-Energy Physics Hardware Design

Dipl.-Ing. Manfred Mücke

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Begutachter: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß
Zweitbegutachter: Dr. Richard Jacobsson (CERN)

Graz, im Juli 2007

Kurzfassung

Die Zeit- und Ortsauflösung sowie die erreichbare Integrationsdichte von Sensorelementen in Hochenergiephysik (HEP)-Experimenten hat sich im Laufe der letzten beiden Jahrzehnte drastisch erhöht. Die von aktuellen Experimenten generierten Datenraten größer 1TB/s, erzwingen neue, hochperformante und über existierende Lösungen hinausgehende Systeme zum Auslesen, Verarbeiten und Speichern der gewonnenen Daten. Field-Programmable Gate Arrays (FPGAs) bieten die Möglichkeit schnelle digitale Signalverarbeitung zu implementieren und erlauben durch Ihre Programmierbarkeit eine rasche Anpassung an geänderte Problemstellungen ohne einen Austausch von Komponenten zu erzwingen. Aufgrund dieser hervorragenden Eigenschaften werden FPGAs heute von zahlreichen Hochenergiephysikexperimenten zur Echtzeitdatenverarbeitung eingesetzt. Dementsprechend ist auch die Bedeutung von Entwurfswerkzeugen zum Umsetzen von Signalverarbeitungsalgorithmen auf FPGAs gewachsen, um die theoretisch erreichbaren Leistungen auch praktisch erzielen zu können.

Diese Arbeit untersucht bestehende Hardwarebeschreibungssprachen auf Ihre Unterstützung von typischen Problemstellungen des Hardwareentwurfes in der Hochenergiephysik, wobei bestehende Mängel aufgezeigt werden. Besonderes Augenmerk wird dabei auf die Möglichkeit und den Aufwand gelegt, Hardwarebeschreibungssprachen um spezifische Sprachkonstrukte bzw. Hilfsmittel zu erweitern. Funktional Sprachen sind besonders geeignet, um entsprechende offene Strukturen effizient zu implementieren. Dies wird anhand von Fallstudien zur Komplexität von HEP-spezifischen Erweiterungen der funktionalen Hardwarebeschreibungssprache HDCaml dokumentiert.

Abstract

Detectors in High-Energy Physics (HEP) have increased tremendously in accuracy, speed and integration. Consequently HEP experiments are confronted with an immense amount of data to be read out, processed and stored. Originally low-level processing has been accomplished in hardware, while more elaborate algorithms have been executed on large computing farms. Field-Programmable Gate Arrays (FPGAs) meet HEP's need for ever higher real-time processing performance by providing programmable yet fast digital logic resources. With the fast move from HEP Digital Signal Processing (DSPing) applications into the domain of FPGAs, related design tools are crucial to realise the potential performance gains.

This work reviews Hardware Description Languages (HDLs) in respect to the special needs present in the HEP digital hardware design process. It is especially concerned with the question, how features outside the scope of mainstream digital hardware design can be implemented efficiently into HDLs. It will argue that functional languages are especially suitable for implementation of domain-specific languages, including HDLs. Case-studies examining the implementation complexity of HEP-specific language extensions to the functional HDCaml HDL will prove the viability of the suggested approach.

Acknowledgements

This work would have been an impossible venture without the help of many people.
The Austrian Doctoral Student Program made my stay at CERN possible. Jorgen was an awesome supervisor, providing advice when needed, but giving freedom generously where ever possible. Together with Manuel and Paulo he dragged me off the office twice a week for some laps in the pool. Thanks for your friendship and for keeping me fit.
Many people at CERN have explained to me their respective area of expertise with endless patience, giving me an idea of the complex environment of a high-energy physics experiment. Guido, Niko, Benjamin and Lars enabled my work and were my comrades-in-arms in endless working hours.
Thanks to Alex ("you really should come to CERN") for initially promoting the option of a PhD at CERN and all the subsequent support. Thanks to Angelika, Gudrun, Kurt, Stefan, Ulrich, Sonia, Peo, Karin, Tomas and many more for their friendship, making life in Geneva so enjoyable.
But the biggest share goes, without doubt, to Anja, my wife. Joining me in this adventure without hesitation, moving to Geneva, and accepting a thousand kilometers between us for quite some time, she made these three years a time we love to look back at.

Graz, July 2007 Manfred Mücke

Extended Abstract

Thanks to a PhD grant, I had the opportunity to work for two and a half years within the LHC beauty (LHCb) collaboration at CERN, the European laboratory for particle physics in Geneva, Switzerland. LHCb is one of the four experiments in preparation for the Large Hadron Collider (LHC) due to be operational by the end of 2007.

This work is based on experience gained by tight cooperation between researchers from distinct domains. It provides insight into how Hardware Description Languages (HDLs) can serve better the specific needs of digital hardware designs in High-Energy Physics (HEP).

Introduction

HEP experiments installed at the LHC generate a tremendous amount of data (for LHCb about 40MS/s * 80KB $\approx$ 3.05TB/s), exceeding streaming capabilities of current storage technologies by orders of magnitude. Consequently large massive-parallel processing facilities are used to select and/or compress data in real-time. Programmable logic (FPGAs) has proved to be a viable technology for implementation of such real-time filters, meeting both the demand of computing power and design flexibility.

Most Electronics Design Automation tools available today consider designs implemented in FPGAs as simple digital hardware, i.e. the typical design elements are wires, basic logic and arithmetic functions, registers and memories. However, mapping HEP algorithms into FPGAs and exploring the respective design space requires a very different approach and notation. This poses a considerable shift of objective, not being met by tools currently available.

This work identifies special design needs of contemporary HEP digital hardware designs. It focuses on the used modelling language (usually a hardware description language) and how to make the relevant model information accessible to tools. Specifically (and in contrast to many other approaches) it considers the potential advantages functional languages can provide. It suggests and demonstrates extensions to HDLs to satisfy typical design needs of HEP experiments.

As the very goal of HEP research is to observe what has not been seen before, operating always at the edge of current technology, many crucial design decisions have to rely heavily on simulations. Once data taking has started and new knowledge begins to take shape, researchers want to refocus detectors to incorporate the new knowledge. Reconfigurable trigger-levels implemented in FPGAs allow - in principle - for such an adaptation without the need to develop new hardware.

However, reality does not live up to the expectations. This is mainly due to limited flexibility provided by the current design flow for FPGAs. Design descriptions at

Register-Transfer Level (RTL) using VHSIC Hardware Description Language (VHDL) do not provide sufficient problem-abstraction to allow fast iteration on algorithm variations (i.e. delivering in short time both simulation and implementation data from a model variant). Therefore every iteration has to be seen as a new design effort (requiring considerable manpower).

Iterating on a trigger implementation means providing a set of alternative implementations for evaluation in software first. The most promising candidate should then replace the current implementation, possibly without affecting the surrounding functions. Experience has shown that turn-around times for such iterations lie within the order of months, if iteration is considered at all. This is due to the difficulty to provide identical models for software simulation and hardware implementation and the range of skills required.

One possibility to improve upon this situation would be to provide:

- a description of algorithms at RTL to guarantee predictable synthesis results;
- a more generic description of algorithms than provided by VHDL to reduce code-complexity and to improve fine-grained design-space exploration;
- automatic generation of fast and bit-accurate simulation models from the same source used for hardware description;
- minimization of the effort required to implement language features enabling a design flow as described above and potential future language extensions.

Objective

No HDL currently available fulfills the requirements listed above. We therefore set out to show that this scenario is nevertheless feasible. We chose an open source HDL allowing RTL description and providing an open type system and means of extending the language. As a proof of concept, we have made the internals of the language more accessible and have implemented different extensions to that HDL. To consider HDL extensions a feasible option for future projects, the amount of work to be invested for implementation is important. We will show that it is low with regard to other similar approaches.

During research of possible language platforms on which our extension could be based, we encountered a lack of suitable extendable traditional HDLs. This led to a more theoretical review of language design principles. We found functional languages much more efficient for domain-specific extensions than traditional imperative ones. While this is in accordance with literature on domain-specific languages, this finding has only rarely been applied to HDLs. We will compare implementation complexity of similar features using both paradigms.

There are many attempts to improve upon the existing HDLs. Most of these approaches however are not publicly accessible. We have the impression that due to the lack of existing language platforms on which investigative approaches can be based, the advances in domain-specific HDLs are moderate to low. We have therefore put emphasis on an open-source approach. The improvements to the used HDL and most extensions have been made

available online[1] and an extensive tutorial has been compiled[2].

Hardware abstraction and problem abstraction are generally orthogonal language features. However, current approaches do not treat related issues independently, or concentrate on hardware abstraction. Our work concentrates on the highest problem-abstraction possible at a given hardware-abstraction level (RTL).

High-Energy Physics compliant simulation models

As access to accelerators is usually limited and expensive, simulation plays a dominant role in preparing and optimizing HEP experiments. The HEP community has created many highly specialized tools or extensions to existing mainstream tools for simulation of very diverse events, ranging from particle-particle interaction to computing farm load distribution. In HEP experiments, algorithms implemented in FPGAs are only a small part of a very big system. Consequently the simulation models have to fit the existing infrastructure rather than the other way round (as is the case in typical Electronics Design Automation environments).

One important language for large-scale simulation is C++ and there exists a varied and highly developed infrastructure of C++-based simulation frameworks. Each of these frameworks requires specific coding conventions to be followed. We have explored HDL language extensions allowing automatic generation of C- and C++-models. We have further investigated the ease with which the generation of code can be tuned such that compliance with specific simulation framework coding conventions can be achieved.

The idea of deriving HEP-specific simulation models from circuit descriptions was presented at the *2005 IEEE Real-Time Conference*[3]. A case study based on algorithms used by the LHCb VErtex LOcator (VELO) sub detector was presented the following year at the *12th Workshop on Electronics for LHC and future Experiments* (LECC'06)[4].

Abstract fixed-point data type

For most trigger algorithms, the Signal-to-Noise Ratio (SNR) is the dominant quality factor. In digital designs, this is equal to the Signal-to-quantization-Noise ratio (SQNR) which is determined by the bit widths chosen for the different data paths. A specific bit width is typically a compromise between precision requirements (derived from physics) and available hardware resources. Physics requirements can however vary a lot over time and a new discovery, a changed trigger algorithm or improved hardware might fundamentally change the respective precision required. It would therefore be favorable to maintain the highest degree of flexibility possible with regard to data bit width over the whole lifetime of a trigger implementation.

---

[1] Tom Hawkins. *HDCaml Home Page*. 2006. URL: `http://www.funhdl.org/wiki/doku.php/hdcaml`.

[2] Daniel Sánchez Parcerisa. *HDCaml Tutorial*. 2006. URL: `http://www.funhdl.org/wiki/doku.php/hdcaml:tutorial`.

[3] Manfred Muecke. "C/VHDL codesign for LHCb VELO zero suppression algorithms". In: *14th IEEE-NPSS Real Time Conference*. 2005. DOI: `10.1109/RTC.2005.1547399`. 156–157.

[4] Manfred Muecke and Tomasz Szumlak. "Unified C/VHDL Model Generation of FPGA-based LHCb VELO algorithms". In: *12th Workshop on Electronics for LHC and future Experiments (LECC'06)*. 2006. URL: `http://cdsweb.cern.ch/record/1034306`.

Consequently we have chosen to implement a HDL-extension which provides an abstract fixed-point data type. Modelling using this data type separates bit width- and precision-considerations from the algorithmic description. It enables generation of design variations with only minimal code changes (due to automatic signal property propagation). This work has been presented at the 2006 *Forum on specification & Design Languages* (FDL'06)[5].

The feasibility of the suggested design flow in combination with the abstract fixed-point data type will be shown by design-space exploration for the LHCb VELO Linear Common Mode Suppression (LCMS) algorithm. The LHCb VELO acLCMS algorithm is a real-world example, and (using a former VHDL model) is implemented in the LHCb Data Acquisition System (DAQ) interface board called TELL1[6], a custom-built multi-FPGA processing board of which 350 are currently being installed in the LHCb cavern.

Graph Description Language model generation

Control and Data Flow Graphs (CDFGs) are a useful intermediate representation when mapping algorithms into sequential circuits and especially powerful when used in combination with graph transformation rules. The Graph Description Language provides a format for exchange of graphs between different applications. We have created a language extension generating Graph Description Language (GDL)-compliant code from the original circuit description. A free tool (aiSee by AbsInt GesmbH) is available for visualization of the generated GDL files. Comparison with similar tools shows a very low complexity required for implementation (due to the direct and efficient access to the language's internal intermediate circuit representation).

Original work

The chosen HDL platform used for the case studies in this work is called HDCaml. It is an open-source project not started by the author of this work, but the language has served us[7] well as a starting point for a number of projects, extensions and investigations. Some of the work has been made available to the public in form of fixes and partial releases. The abstract fixed-point data type, C++ model generator and CDFG extraction are genuine work conducted by the author.

The scientific contribution of this work consists of:

- an extension to the HDCaml HDL providing automatic generation of bit-true C++ simulation models. The resulting code can easily be tuned to comply with different coding conventions.
- an extension to the HDCaml HDL to represent abstract fixed-point data types, providing a notation such that manual code-intervention is minimized when performing

[5] Manfred Muecke and Guido Haefeli. "A bitwidth-aware extension to the HDCaml Hardware Description Language". In: *Proceedings of the Forum on specification & Design Languages 2006, FDL'06*. ECSI, 2006.

[6] G. Haefeli et al. "The LHCb DAQ interface board TELL1". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 560.2 (May 2006). 494–502. DOI: 10.1016/j.nima.2005.12.212.

[7] Jean-Philippe Brantut(CERN), Daniel Sánches Parcerisa (CERN), Reinhold Schmidt (ITI), Stefan Lickl (ITI), Karl Flicker(ITI)

bitwidth-related modifications to the design.

- showing that functional languages lend themselves to a very efficient implementation of such a language extension, thereby furthering the case of using functional languages for implementation of domain-specific hardware description languages.
- a module allowing extraction and export of a corresponding control and dataflow-graph, thereby complementing similar efforts for other RTL-HDLs. Comparison shows a very low complexity of the required coding.

Contents

List of abbreviations

ASIC	Application-Specific Integrated Circuit
CDFG	Control and Data Flow Graph
CERN	Conseil Européen pour la Recherche Nucléaire
DAQ	Data Acquisition System
DSL	Domain-Specific Language
DSP	Digital Signal Processor
DSPing	Digital Signal Processing
EDA	Electronics Design Automation
FPGA	Field-Programmable Gate Array
GDL	Graph Description Language
GPP	General-Purpose Processor
HEP	High-Energy Physics
HDCaml	Hardware Description Objective Caml
HDL	Hardware Description Language
IDE	Integrated Development Environment
LHC	Large Hadron Collider
LHCb	LHC beauty (experiment)
LCMS	Linear Common Mode Suppression
OCaml	Objective Caml (functional programming language)
RTL	Register-Transfer Level
SoC	System-on-Chip
VELO	VErtex LOcator
VHDL	VHSIC (Very-High-Speed Integrated Circuit) Hardware Description Language

Chapter 1

Introduction

The aim of this work is to explore how Electronics Design Automation (EDA) tools can be tuned to fulfill better the needs of advanced digital designs in High-Energy Physics (HEP) experiments. Computer languages serving a specific community are called domain-specific languages. We extend this nomenclature to Hardware Description Language (HDL), calling HDLs tuned toward the needs of a specific community **domain-specific HDLs**. While the suggested specialisation and presented case studies are motivated by HEP designs, the findings on how to extend HDLs apply without restriction.

Motivation

HDLs currently used in industry, like VHDL and Verilog, have been originally designed as means to describe already existing electronic circuits for the purpose of documentation. Over time, and due to advances in EDA software, HDLs have become the dominant tool for design entry. A typical design flow nowadays uses the same HDL source code for circuit specification at register-transfer level (RTL), simulation/verification and implementation. This results in a unified design process.

Additionally, circuit size of designs has grown exponentially over time due to the ever decreasing feature size of modern integrated circuits. To cope with the increasing design size, there is a continuous need to increase the level of abstraction used for circuit design. As circuits are mainly designed using HDLs, this implies increasing expressiveness of HDLs. Current HDLs fail to provide adequate levels of abstraction for an increasing number of designs.

The research community has been investing much work in higher-level synthesis (also called behavioural synthesis), which aims at synthesizing hardware from a more abstract description, usually above the register-transfer-level (RTL). While such an approach can lead to much shorter and more versatile descriptions, the final synthesis result is often unpredictable. High-level synthesis is an active area of research.

Therefore, by picking a language for circuit design entry, one has currently to choose between

- hardware-centric (low-level) descriptions, which lack problem abstraction, or
- algorithmic (high-level) descriptions where resulting hardware is hard to predict.

The latter is especially unfavorable for designs whose speed or resource requirements approach the limit of the target platform.

Consequently, for complex digital designs, challenging todays hardware platforms, there exists no HDL satisfying the needs of a unified design process.

The design of circuits for digital signal processing (DSP) applications has suffered from this fact ever since integrated circuits have become powerful enough to be used for DSP applications. Consequently the industry has come up with a plethora of design aids, among which *C-to-Hardware* approaches have been especially prominent in the last years. Most approaches allow for some kind of higher-level problem description and try to compile suitable hardware, often directed by design directives given by the designer. While this increased abstraction enables fast design-space exploration, it also reduces the predictability of generated hardware. This is of little concern to applications being moved from much slower platforms (like moving functionality originally implemented in software into hardware). However, it renders these approaches unattractive for design iterations requiring predictable results. Predictable results are mandatory for fast evaluation of fine-grained variations, typical for high-speed applications.

To improve upon this situation, better (i.e. more abstract) specification languages together with predictable transformation rules to derive actual hardware (higher-level synthesis) are required. One possible approach is to incrementally extend currently available HDLs to test effectiveness of experimental language features. Every effort to create, extend or modify a HDL however, has to take into consideration not only language design issues but also the respective implementation effort. This effort becomes the more important the smaller the respective user base becomes. A given effort might be acceptable for an expected large user base, but might be prohibitive for features requested only by a small community. As HDLs in general have a very limited user base (compared e.g. to general-purpose computer programming languages) the effort required is of serious importance.

To facilitate future evaluation of domain-specific language extensions, an extendable HDL platform is required. This work will contribute to these efforts by identifying possible platform candidates and by presenting case studies of languages extensions. Specifically it will elaborate on the potential benefits functional languages can provide to such an extendable HDL platform.

Document Structure

Chapter 2 gives a general overview on HDLs, their history, their place in the hardware design flow and typical levels of abstractions employed. Basic concepts of domain-specific languages (DSLs) are introduced and discussed. Finally experimental HDLs are reviewed both from a HDL and DSL point of view, outlining motivations for additional language features and respective implementation techniques.

Chapter 3 discusses HDLs from a perspective of hardware design in HEP experiments. Specific requirements will be identified and shortcomings of current approaches will be outlined. This discussion will be supported by a review of recently developed designs and respective design methodologies.

Chapter 4 identifies hardware design needs in HEP not shared with other application domains (HEP-specific needs). Based on these HEP-specific needs, new language and tool

features are suggested to enable an improved HEP hardware design methodology.

Chapter 5 will present case studies of HDL-specific tool and language feature implementations, relevant to needs identified in the preceding chapter. For all three case studies, existing solutions and tools will be discussed. The HDCaml HDL will serve as a testbed for the presented case studies. The case studies will

1. present an extension to the HDCaml C-model output generator, enabling mapping of existing models into C++ models conforming to specific coding conventions.

2. present an extension to the HDCaml HDL allowing flexible fixed-point models.

3. present a new HDCaml output generator, enabling mapping of existing models into the graph description language (GDL).

Chapter 6 will summarize the work and will discuss possible future directions of research.

Appendix A gives technical details of the HDCaml implementation, referred to throughout chapter 5.

Chapter 2

Hardware Description Languages (HDLs)

> "A language encapsulates the complete knowledge of all processes within its scope of modeling."
>
> "The very goal of any HDL is to enable a designer to express a desired functionality, such that it matches the behaviour of an existing or yet-to-be-build hardware system."
>
> Sumit Ghosh [Gho99]

HDLs provide a formal framework for modelling electronic circuits in a computer-readable manner. Formal specifications enable partial automation of the design process, thereby reducing its complexity. This in turn improves productivity, which is a key requirement to cope with continuously increasing design sizes.
VanCleemput lists the three major areas of applications for HDLs [van77]:

1. Documentation of a design's specification
   - to allow communication between designers and users and
   - to serve as a tool for teaching hardware design and computer architecture.
2. Providing input to the system level simulator (Simulation).
3. Providing input to the automatic hardware compiler or synthesizer (Implementation).

HDLs exist for modelling analogue systems, digital systems and a combination thereof (mixed-signal systems). This work's concern is with digital systems and observations will therefore be restricted to HDLs targeting digital systems.

Understanding the development of HDLs, their features, deficiencies and cost of implementation, is the key to applying them effectively to future challenges in digital hardware design. In the following, the history of HDLs, their place in the design flow and the types and levels of abstraction they provide will be reviewed. I will emphasize the "abstraction gap" and will present some experimental HDLs. Finally I will elaborate on the process of creating formal languages with features tuned specifically to the requirements of a given application domain (domain-specific languages).

2.1 History

HDLs emerged from simple tools for automating repetitive tasks in early circuit design. These tools were followed by tools allowing simulation of a set of interconnected logic gates (gate-level simulators). The languages defining the input to these tools can be considered the first HDLs. Examples are LAMP[Cha76] (Logic Analyzer and Maintenance Planning), SALOGS[CS78] (Sandia Logic Simulation System) and TEGAS[SL73] (Test Generation and Simulation System).

Following the increasing design size, new tools appeared during the late 1960s, allowing more succinct formulation of models at a higher level of abstraction, namely at register-transfer level (RTL). Examples of such RTL HDLs are CDL[Chu65] (Computer Design Language), DDL[DD68] (Digital System Design Language) and AHPL[HP87] (A Hardware Programming Language).

All HDLs mentioned so far modeled digital combinational or synchronous logic. Truly asynchronous (uncoordinated in time) systems could only be modeled with the advent of the first behaviour-level HDL: ADLIB[Hv79] (A Design Language for Indicating Behavior). ADLIB is a superset of PASCAL with special facilities for modelling concurrency and interprocess communication. It is normally used under the SABLE simulation system.

Ghosh[Gho99, p.19] defines behavioural-level HDLs as being capable of modelling both synchronous and asynchronous system at any level of abstraction. He points out[Gho99, p.15] that the term has been often misused in literature as a synonym for "architectural-level". A behavioural model is therefore one in which different, potentially not synchronized, models at different levels of abstraction (gate-level, register-transfer level, architectural level, ..) can coexist.

SDL[van77] (Structural Description Language) complemented ADLIB as a language to express structure and connectivity of a digital design at arbitrary levels of abstraction. It featured a macro expansion syntax to support manual refinement of designs.

Building upon the experience gained with ADLIB and in view of expected future very high speed integrated circuits (VHSIC), the United States Department of Defense (DoD) decided to coordinate development of a new HDL. It outlined the requirements in its Draft Request for Proposal[Dep82], DRFP F33615-83-R-1003, published in September 1982. The language was named **VHDL** (Very high speed integrated circuits Hardware Description Language) and the DRFP requested that it was to adhere to the Ada programming language syntax wherever possible. Coordination of VHDL was later transferred from the DoD to the Institute of Electrical and Electronics Engineers (IEEE) and VHDL was adopted in form of the IEEE Standard 1076[IEE88], *Standard VHDL Language Reference Manual*, in 1987 (often referred to as VHDL-87). In 1992, a revised version was proposed and finally accepted in 1993, leading to VHDL-93[IEE94]. Another revision in 2001 led to VHDL-2002[IEE02], which is the current version as of time of writing (2007).

The **Verilog** HDLs was invented by Phil Moorby at Automated Integrated Design Systems in 1985. Automated Integrated Design Systems was later renamed to Gateway Design Automation and purchased by Cadence Design Systems in 1990. Cadence transferred coordination of the design effort to the Open Verilog International (OVI) organization. OVI submitted Verilog to the IEEE which led to acceptance of Verilog as IEEE Standard 1364-1995, commonly referred to as Verilog-95. A revised version was published in 2001 as IEEE Standard 1364-2001 (Verilog 2001). The latest revision dates from 2005, pub-

lished as IEEE Standard 1364-2005 (Verilog 2005). Verilog's syntax borrows from the C programming language, but conceptually provides the same modelling features as VHDL. Compared to the more elaborate VHDL, Verilog offers more compact model descriptions. Ghosh[Gho99, p.63] points out however that "it lacks a number of important characteristics that may cause the generation of erroneous results and hinder the correct representation of hardware".

Verilog and VHDL have both achieved a wide acceptance and have together become the de-facto standard for digital system design. There are efforts to extend both languages' scope to modelling analogue and mixed-signal (AMS) systems (VHDL-AMS, Verilog-AMS).

With the advent of Systems-on-Chip (SoC) at the end of the 1990s, i.e. the possibility to integrate both processors and dedicated hardware units on a single chip, *architectural exploration* (evaluating tradeoffs of varying high-level parameters in the construction of a SoC) gained importance. To be able to evaluate impact of design decisions, a more unified simulation environment became necessary, allowing coexistence from software and hardware at different levels of abstraction. SystemC and SystemVerilog were designed as a response to these needs.

SystemC is a "C++ class library for system and hardware design for use by designers and architects who need to address complex systems that are a hybrid between hardware and software"[IEE06]. The libraries contain a simulation kernel, which enables execution of SystemC models. SystemC therefore is both a modelling language and a simulation environment. SystemC allows separate description and iterative refinement of modules and communication between these modules. It provides the means to describe a modules' functionality and communication as applicable to digital systems. It enables a self-contained system description at different levels of abstraction, comprising both hardware and software (expressed in C++).

SystemC was originally invented by Cadence, Inc. during the 1990s. To enable wider acceptance, SystemC was released as open source and coordination was transferred to the Open SystemC Initiative (OVI) in 1999. SystemC was accepted by the IEEE as Standard 1666-2005[IEE06] in 2005.

While SystemC serves well the tasks of documentation and simulation, its usefulness to synthesis (both practical and theoretical) is debatable. There is an ongoing academic argument on the suitability of C-like languages (SystemC being a prominent example) as input to hardware synthesis [Edw06]. The SystemC standard itself neither mentions hardware synthesis as a primary target, nor does it specify a synthesizeable subset of the language (while there exists an effort to agree on a synthesizeable SystemC subset, as of time of writing there exists only a draft [OSC04], which seems to make little progress since its appearance in 2004). It remains to be seen to what extent the iterative refinement of designs, as suggested by SystemC, can be automated by external tools.

Design Flow

HDLs are tools invented to support the hardware design process. To better understand requirements and limitations of current HDLs, we review the hardware design process and point out links to HDLs.

Hardware design usually follows a

1. specification-
2. model construction/implementation-
3. verification/simulation-
4. optimization-
5. refinement-

iteration cycle at different levels of abstraction. From a given Specification, a model is constructed. Using this model, the functionality is verified by suitable tests at the respective level of abstraction. If the model meets the functional requirements, it can be optimized to meet specific design goals. The resulting description then serves as specification to the next cycle at a lower level of abstraction (refinement). The lowest-level model will result in actual hardware, complying to the requirements at all levels of abstraction.

A popular model to express these stages of design is the V-model. The picture shown in Figure 2.1 describes the basic development flow in hardware design. The process starts by providing system-level specifications and refines them, while going down the first leg of the 'V', capturing requirements, deriving more detailed specifications and architecture. At each stage tests are developed and applied to the refined models.

During the upward path of the 'V', actual system integration is being performed with actual prototype components and tested in their respective environment. Arriving at the highest level, the result should be a fully assembled system, consisting of tested subcomponents, complying to specifications on all levels of abstraction.

Development of tests and initial verification of the models should take place in parallel with the specification or refinement at the respective level of abstraction. This specifically means that tests at all levels of abstraction should be available before actual hardware implementations exist.

It is recognized that the quality of the design specification is one of the most decisive elements in the success of the design. Suitable test design is the most cost-effective precaution to detect design flaws early in the design process.

We will give a short overview on levels of abstraction and refer to chapter 2.2 for full details. Hardware design starts with a *system-level* specification, which is then optimized for some given boundary conditions. Taking the optimization into account, the design is refined (implemented at the next lower level of abstraction) and verified to be consistent with the original specification. If the system-level description lacks synchronization (coordination in time), it is identical with a behavioural description.

Module-level specification splits functionality into blocks corresponding to single physical entities. System-on-Chip allows skipping of this step as all functional blocks are implemented on the same chip.

At *architectural-level*, basic functional units on a chip and their interaction is defined.

At *algorithmic-level*, a part of the design is specified by a defined sequence of steps (algorithm). This can be done with or without explicit notion of hardware.

Register-transfer-level (RTL) models describe functionality as a set of synchronized memory elements (registers), and combinational logic, connecting them.

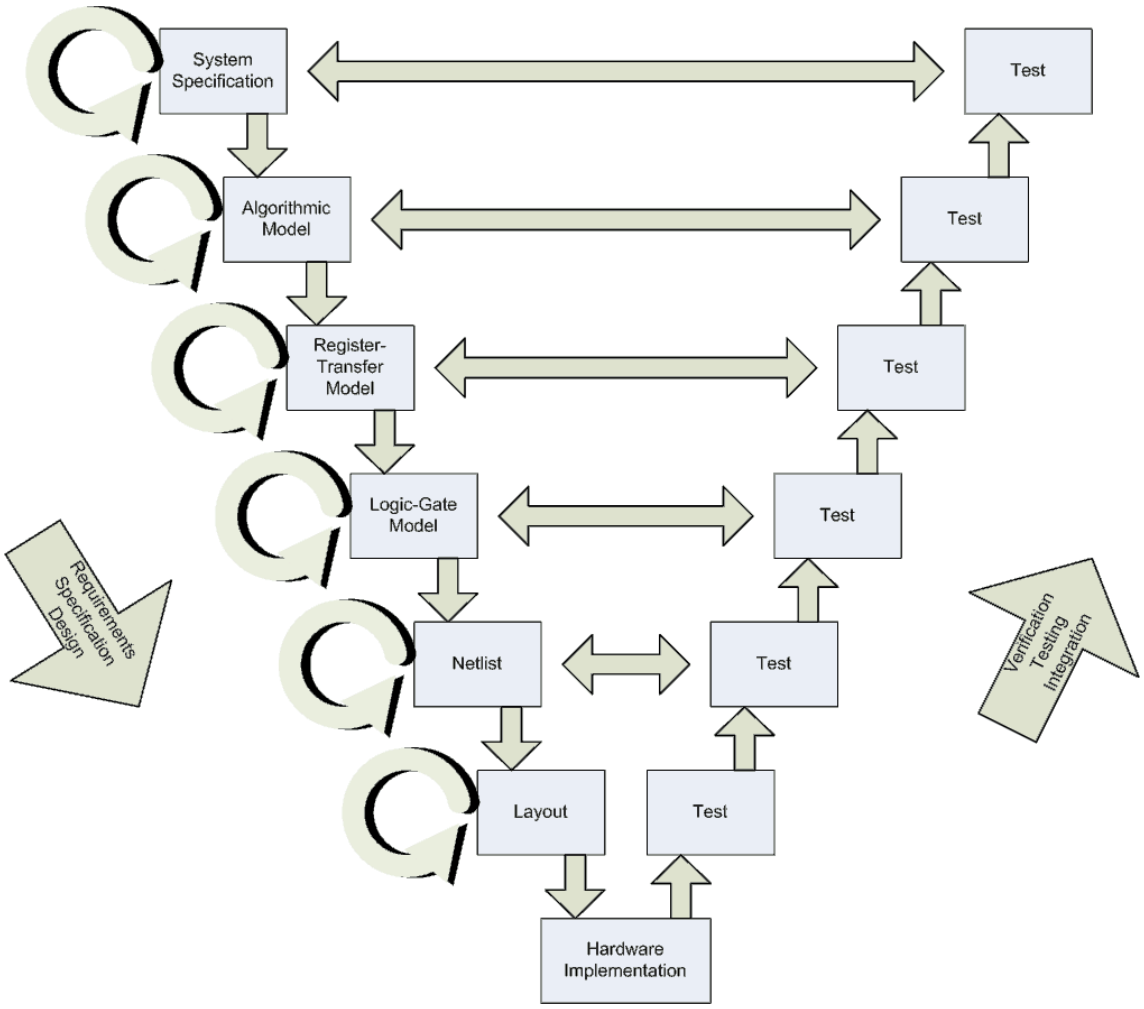


Figure 2.1: V-model depicting phases of the hardware design process

Gate-level models describe functionality as a set of interconnected basic logic gates.

Transistor-level models describe functionality as a set of interconnected transistors.

Electronics Design Automation (EDA) software may support hardware designers at any of the tasks described above. The preceding chapter has shown how HDLs have started from the lowest level and continuously supported higher levels of abstraction. Today, RTL is the generally accepted level of abstraction for design entry of hardware designs, where tools can be expected to support all necessary tasks down to hardware implementation.

A typical design-flow for single-chip systems would therefore comprise specification, verification, implementation/simulation, optimization, and refinement at system-, architecture- and register-transfer-level.

What jeopardizes this idealistic top-down design approach are existing building blocks (intellectual property, IP) at different levels of abstraction and incomplete information to accomplish optimization at a given level of abstraction.

- Existing building blocks introduce additional limitations to the optimization process,

as block boundaries at the level of the IP have to be accepted. An existing IP-block at RTL i.e. can be simulated at architectural-level, but no functionality can be included or stripped.

- Optimization of a given functionality usually requires stepping down several layers of abstraction (sample implementation) before the necessary data to evaluate the requested cost-function is available. Achievable clock frequencies, logic resource usage or heat dissipation can usually only be derived accurately from a fixed (after place and route) gate-level description. This sample implementation should in principle span the whole range of possible implementations. The implied complexity and required effort are usually prohibitive however. This is often circumvented by replacing (time-consuming) sample implementations by (fast but imprecise) optimistic and pessimistic estimates of selected design key figures. While the figures might be key requirements to high-level optimization, simulation will deliver inherently (more or less) unreliable results.

Most real design processes are therefore a mixture between bottom-up and top-down approaches and *interoperability* between the models at different levels of abstraction is one of the dominant factors to the overall achievable design performance. It also defines achievable implementation correctness as only automated tests, spanning all levels of abstraction, can guarantee compliance with original specification. The more layers of abstraction a test-suite comprises, the more trustworthy will its results from a system-verification point of view be.

2.2 Design Representation

Depending on the needs of different users, the same design can be described by using different descriptions, or *design representations*. Different design representation therefore present a different view on a given design. They do not alter the design. Depending on the suitability of the representation, different tasks might however be easier or more difficult to accomplish.

Gajski and Kuhn [GK83] have identified the three most common types of design representations as: functional, structural and physical representation.

- **Functional representation** defines the response of a functional block as a function of its inputs and expired time. Whatever description is used, it has no relation to the actual implementation inside the functional block (black box). Computer-readable functional representation is the classical domain of HDLs.

- **Structural representation** describes a system in terms of basic building blocks, instances thereof and interconnects (wires). The computer-readable form of a structural representation is called a *netlist*.

- **Physical representation** contains a detailed description of the physical characteristics of each building block including dimensions, locations relative to each other and the connections given in the structural representation. The physical representation typically used in chip design is the *layout*, from which the masks for integrated circuit production are derived.

The Y-chart introduced by Gajski and Kuhn [GK83] depicts these three representation by using three axes, forming the characteristic "Y" (see figure 2.2). The circles connecting specific representations at same distance from the center, show distinct levels of abstraction.

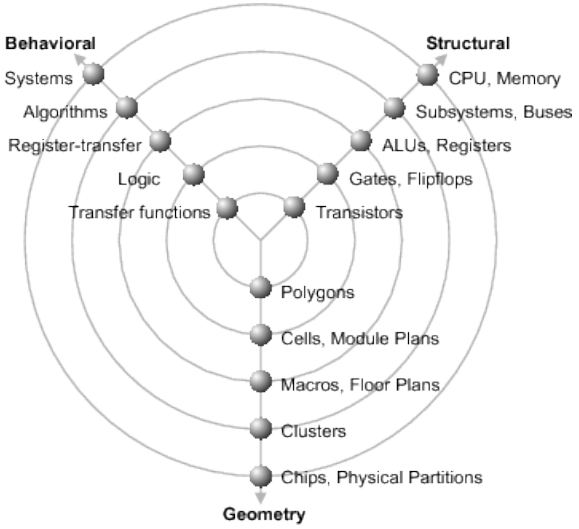


Figure 2.2: Y-chart depicting different levels of abstractions and design representations in hardware design (pursuant to [GK83]).

While the different representations are equivalent in the Y-chart, different representations are typically more or less suited for the dominant tasks at a given level of abstraction. Figure 2.3, showing examples of dominant representations at different levels of abstraction is therefore less precise but more intuitive.

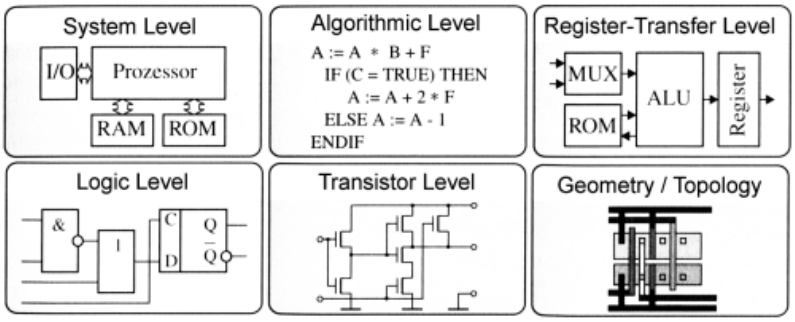


Figure 2.3: Examples of dominant design representations at different levels of abstraction in hardware design [Sch01].

As this work aims at improving the design process by HDLs features at Register-Transfer Level (RTL) (HDL models at RTL and above being a functional design view), functional representations will be emphasized in the following. This is however not limiting the importance of structural and physical representations for other design tasks.

Types and Levels of Abstraction

> Abstraction is the act of considering something as a general quality or characteristic, apart from concrete realities, specific objects or actual instances.
>
> Webster English Dictionary[Web94]

Abstraction typically results in complexity reduction, leading to a simpler conceptualization of a domain in order to facilitate processing or understanding of many specific scenarios in a generic way.

In the simplest case, the desired functionality can immediately be expressed by the designer in terms of available basic building blocks and necessary interconnects. The language only needs to support constructs and data types matching exactly the target hardware's capabilities. It acts merely as a tool for documentation and verification/simulation.

The more complex the desired functionality however, the more unlikely it becomes that a designer will be able to map it directly into basic building blocks. Increasing the level of abstraction is one of the driving forces behind development of new HDLs as design sizes increase continuously.

The task of *refinement*, i.e. mapping a given specification into a model at a lower level of abstraction is one of the native tasks of a system designer. Formal Languages supporting this process should provide constructs and data types matching the designers need at the relevant level of abstraction. Languages without direct implementation are called *Specification Languages*. A HDL is a (specification) language at a level of abstraction low enough to allow automatic implementation of the described circuit by a tool (hardware compiler, synthesizer).

The difference in level of abstraction between the used pure specification language and the used HDL determines the effort required by the designer. This difference in abstraction has been found to be important in contemporary hardware design.

HDLs can be characterized by the *level of functional abstraction* they provide. If the considered HDLs target similar hardware (digital logic in our case), the level of functional abstraction can be compared easily between different languages and serves well as a common classificator.

Domain-specificity determines how natural one can express a specific problem in the given HDL. HDLs do usually not match very well the original problem domain. It is also very difficult to compare different HDLs against each other as the range of problem domains is wide and increases continuously with digital logic penetrating more and more applications. I will motivate in the following why domain-specificity should be considered a vital feature of HDLs and that hardware-abstraction and domain-specificity are not necessarily correlated.

Functional Abstraction of Hardware

Functional abstraction of hardware (hardware abstraction) allows description of a circuit (or subcircuits) by specifying its response to given input values over time. Gajski's and Kuhn's Y-chart [GK83] shows increasing levels of abstractions along its axes (see figure 2.2). The levels of hardware abstraction can be found along its axes for functional representation. In the following we give an overview on the taxonomy of levels of hardware abstraction, as outlined in [Gaj96]:

- *Circuit level:* At the circuit level, a system is modeled using the basic elements of an electric circuit (resistor, capacitor, inductor).

- *Transistor level:* At the transistor level, a system is modeled in terms of transistors. As transistors are the dominant basic building blocks of all digital systems, this level of abstraction has gained special importance in design of digital system. It is also the level of abstraction differentiating digital systems from other electronic systems (which can usually not be modeled using transistor-level models).

- *Gate level:* At the gate level, a system is modeled in terms of digital logic gates, implementing basic boolean operations (and, or, nor, xor) on binary signals.

- *Register-transfer level:* At the register-transfer level (RTL), a specification is given in terms of interconnected combinational logic and storage elements (registers). While the timing and dataflow is fixed, there is still freedom to the implementation of the combinational logic itself.

- *Architectural level:* The architectural level borrows its meaning from *computer architecture*, the science of the conceptual design and fundamental operational structure of a computer system. At the architectural level, a system is modeled in terms of basic building blocks like memories, central processing units (CPUs), memory management units (MMUs), coprocessors, busses and switches interconnecting different functional blocks and inputs/outputs (I/Os).

- *Algorithmic level:* At algorithmic level a part of the design is specified by a defined sequence of steps (algorithm). This can be done with or without explicit notion of hardware.

  Expressing algorithms in a mathematical notation (no reference to hardware) is most often encountered in signal processing systems which are built around algorithms. For ease of simulation, algorithmic models are often expressed in executable languages (MatLab, C, C++).

  Expressing algorithms in terms of basic commands of a computer programming language (abstract references to basic abilities of a processor and memory model) can often be encountered in control-dominant applications. Such algorithms can often be mapped onto processors embedded in the design.

  System-level languages aim at providing means to model both data-flow- and control-flow-dominant parts of a design in one common environment.

Domain-Specificity

Domain-specificity determines how natural one can express a specific problem in a given language. Natural language is usually ambiguous and professionals of different fields tend to establish a terminology allowing them to unambiguously describe a problem in few words. Mathematical notation is an example of such a domain-specific terminology.

In computer science, domain-specific programming languages (DSLs) trade generality for domain-specific language features. The use of domain-specific features usually enables

improved analysis, verification, optimization, parallelization and transformation in terms of language constructs [MHS05].

There are several ways to deliver domain-specificity:

- *Domain-specific notations* increase productivity by allowing experts in the respective domain to express their knowledge in a native way. In hardware design e.g., time, a concept foreign to most computer programming languages, is of paramount importance. VHDL e.g. features a data type `time`, related units (`hr, min, .., fs`) and a `wait` statement, enabling modelling of delays in a natural and accurate way (`wait for 50ns;`).

  In terms of logic gates, multiplication is a very complex operation. Yet all HDLs today provide an operator '*', allowing abstract formulation of this mathematical operation enabling designers to express functionality in a familiar (mathematical) notation.

- *Domain-specific constructs and abstractions* map ideas important to the problem domain into language features.

  VHDL for example defines its signal assignment operation to be an atomic parallel action: `a <= b; c <= d;` describes two parallel actions, independent of their relative location in the code (in contrast to most computer programming languages, where higher line count implies later execution in time).

  The provision of relevant constructs and abstractions can - depending on the problem domain - pose a major challenge and might require design of a language from ground up.

Hardware Abstraction vs. Domain Specificity

The more constructs and data types, matching the needs of the original problem specification, a HDL provides, the easier the mapping from a specification language into the HDL. Ideally, specification language and HDL can be merged. Automatic refinement to actual hardware (*higher-level synthesis*) however becomes usually the more difficult, the more abstract the original specification.

There are two ways to deal with this situation:

- One way aims at providing a powerful, single environment for specification and refinement, accepting that automatic refinement becomes a very complex problem or that refinement is defined as a manual task. In cases where automatic refinement can be implemented, this solution offers the most design alternatives and potentially enables finding of an optimum solution.

- The other way makes a clear distinction between specification language and implementation language, allowing a more narrow feature set to the implementation language. Design decisions, requiring knowledge in between the lowest abstraction layer of the specification language and the highest one of the implementation language, have to be taken by the designer. Automatic refinement based on input formulated in the implementation language can not revert these design decisions, leading to a more restricted design space. One trades ease of tool implementation for global optimization.

The question is: Where to draw the line? To understand better potential benefits and burdens, it is important to understand implications of different language features and abstractions. I will therefore review language features and abstractions identified in the past by reviewing the literature on experimental HDLs, each targeting a specific shortcoming of earlier approaches.

2.3 Domain-specific Languages

Domain-specific languages (DSLs) often emerge from experience and gradual discovery of deficiencies in initially used general-purpose programming language (GPL) for a specific domain. While GPLs and corresponding tools (compilers, ..) are designed by professional computer scientists, trained in language design and compiler construction, the opposite is often true for DSLs. Modelling deficiencies are usually discovered by users in the respective domain which most probably lack specific knowledge in language design and compiler construction. As Mernik et al. point out in their recent review on DSL development [MHS05], this might be the main reason why literature on DSLs design and construction is very scattered. Consequently many projects end up using an ad-hoc approach neglecting the (difficult to access) body of already existing work.

This is even more true in the field of hardware design. From a computer science point of view, HDLs are already domain-specific languages (for hardware description). The user base of HDLs is much smaller than the one of GPLs and rarely extensively trained in computer science, therefore lacking the theoretical foundations necessary to judge different programming languages' features. Hardware designers usually accept HDLs as a necessary mean to cope with their design, but do not attempt to question the language itself. This is reinforced by the effort necessary to provide efficient hardware synthesizers, the difficult adaptation to devices of different vendors (in the case of FPGAs) or process technologies of different production facilities (in the case of ASICs) and a wide-spread closed-source (an even no-publish) approach present in the EDA industry.

These observations are in accordance with [Vah03], encouraging "hardware designers to become better programmers" and demanding that "curriculum designers must fundamentally rethink the introduction of programming and digital design to new engineering and computer science students"

Maybe this situation will improve over time as the computer engineering curriculum suggested jointly by IEEE and ACM [McG03] identifies programming language fundamentals as a core knowledge, listing specifically: fundamental programming constructs, problem solving and data structures, programming paradigms, recursion, event-driven and concurrent programming, and using application program interfaces (APIs).

In the following, some fundamentals on Domain-Specific Language (DSL) analysis, design and construction will be reviewed. A taxonomy taken from [MHS05] will be used to classify DSLs. Programming paradigms and their impact on hardware design will be discussed.

Analysis

DSL analysis is the task of identifying and understanding the problem domain to such an extent, that relevant notations, abstractions and language features can be derived. This

step requires intimate understanding of the problem domain, language design and the target application (it is important to remember that in hardware design, the target application is usually at least threefold: documentation, simulation and implementation). In contrast to computer languages, conforming to existing notations is paramount in DSLs, as it is the very motivation of a DSL to serve better the needs of a highly specialized community (not necessarily being trained programmers). The documentation of identified notations and abstractions should be as precise and rigid as possible to foster a clear understanding of the aim of the DSL and to avoid inconsistencies and undesired implications in succeeding DSL design and implementation.

Design

DSL Design is the task of linking required notation and abstractions as identified in DSL analysis with implementable language features.

Mernik et al. [MHS05] list the following DSL design patterns, classifying the relationship between a DSL and existing languages:

- *Language invention* describes a DSL without any relation to existing languages. This is a very rare case.
- *Language exploitation* takes to some degree advantage of an existing languages, by any of the following means:
  - *Piggybacking* tunes existing features of a language to provide domain-specific functionality. This alters the functionality of the original language.
  - *Specialization* restricts an existing language to focus it on the targeted application domain. This reduces the functionality of the original language.
  - *Extension* uses an existing language and extends it by adding functions, data types or notation. This extends the functionality of the original language, safeguarding all of its original features (in contrast to piggybacking, where domain-specific functionality overlays and replaces original language features).

The language specification can be given in an informal or a formal notation. Informal notation usually encompasses some natural language description together with code examples. Formal notation can take advantage of existing semantic definition methods and related tools (see [SK95] for a recent reveview).

Implementation

Implementing a DSL is a major undertaking. Implementation decisions and related consequences should be considered carefully. When reviewing DSLs from a common domain, having a sound understanding of implementation principles enables identification of possible paths for adoption and extension of existing languages.

Mernik et al. [MHS05] list the following basic DSL implementation principles:

- Interpreter: DSL constructs are recognized and interpreted using a standard fetch-decode-execute cycle. This approach is appropriate for languages having a dynamic character or if execution speed is not an issue.

- Compiler: DSL constructs are translated to base language constructs and library calls. A complete static analysis can be done on the DSL program. DSL compilers are often called *application generators*.
- Preprocessor: DSL constructs are translated to constructs in an existing language (the *base language*). Static analysis is limited to that done by the base language processor.
- Embedding: DSL constructs are embedded in an existing GPL (the *host language*) by defining new abstract data types and operators. Application libraries are the basic form of embedding.
- Extensible compiler/interpreter: A GPL compiler/interpreter is extended with domain-specific optimization rules and/or domain-specific code generation. While interpreters are usually relatively easy to extend, extending compilers is hard unless they were designed with extension in mind.
- Commercial Off-The-Shelf (COTS): Existing tools and/or notations are applied to a specific domain.
- Hybrid: A combination of the above approaches.

Programming Paradigm

The two dominant programming paradigms are imperative and functional programming [MA01]. Choosing a programming paradigm is probably the most basic decision to be taken when designing a DSL.

Imperative programming is characterized by programs holding a state and commands depending on and modifying the current state. An imperative construct is defined as one changing an already existing value[MA01]. The C-command `x = x+1` overwrites the initial value at the memory position `x` refers to.

Functional programming is characterized by the fact that most computation is done by evaluation of expressions that contain functions[MA01]. Functions calculate a result depending on a set of input values. If a function's result depends exclusively on the specified input values, it is called *side-effect free*[MA01]. Languages providing side-effect free constructs only, are called *pure functional languages*[MA01].

The language paradigm chosen for a DSL influences all areas of the language, but probably the following three ones the most:

1. Language design: Imperative and functional languages have a different "look-and-feel". Independent of the original reason to choose a language paradigm, one should be aware of the implicit consequences.
2. Language implementation: Some applications qualify more or less for a specific programming paradigm. Implementing an application with a less suitable language will result in increased implementation effort and maintenance cost. There is a strong opinion in the research community that functional languages are superior to imperative ones for implementing compilers and embedded languages [MHS05, p.330].

3. User base: The more a new language differs from language concepts common to most predecessors in the respective application domain, the more difficult it becomes to convince prospective users to evaluate it. Consequently the initial user base will be small and growth will not necessarily correlate with the quality of the solution provided.

The paradigm chosen in language design and in language implementation can differ. It is perfectly possible to implement an imperative programming language using a functional language and vice versa. When exploitation is used as a design pattern however, the paradigm of the language chosen for implementation will be visible in the language itself.

Imperative vs. Functional Programming in Hardware Design

It is worth noting that all HDLs in use in industry today are imperative ones. Are therefore imperative languages better suited for modelling hardware?

Low-level programming of von-Neuman architectures leads almost automatically to imperative programming. A single instance (the CPU) being in a specific state (the contents of all memory locations and registers) reads data from memory, modifies it and writes it back to memory (changes the state). Because memory is a limited resource, values are usually overwritten, making them disappear after usage. This has led to C (designed for efficient low-level programming of computers) adopting a very strong imperative programming paradigm.

Because results of expressions in imperative programming potentially rely not only on the given input values but also on the current state, they encourage a sequential execution style. Only if the sequence of execution of a given set of commands is known, the result can be predicted. Therefore explicit execution order is crucial in imperative programming. Extracting parallelism (independent sequences of commands) from imperative programs therefore requires thorough analysis and is an active area of research [SSDM01]. It is a matter of controversial debate, to what extent original algorithmic parallelism can be recovered by analysis of sequential code [SSDM01; Edw06].

Hardware on the other hand is inherently parallel and exploiting this parallelism is one of the prominent tasks of hardware designers to achieve improved computing performance. Parallelism at different levels of granularity can easily be extracted from functional programs [MA01].

There is actually a strong relation between the domain of parallel programming and functional programming languages[Szy94].

The answer to the initial question requires defining how important *automatic* extraction of parallelism is for a given model. Clearly this is of low importance in structural models, because the parallelism is explicitly stated by the designer (in hardware modelling, declaration of several instances of a building block usually implies that they are executed concurrently). The more abstract a functional description becomes however, the more important it is to identify potential parallelism to achieve optimal implementation in hardware. A suitable programming language allows automatic exploitation while the task of providing variants of the design is left to the designer otherwise. It becomes clear from the above that the more abstract models of digital hardware become, the more the adoption of a functional programming paradigm will become an enabling factor for design-space exploration.

There remain however two important questions to be answered:

- Can all tasks important to HDL designers be mapped efficiently into functional HDLs?
- Are hardware designers willing to learn a new HDL requiring not only knowledge of new syntax but also adoption of a new programming paradigm?

The first question can probably only be answered after broad evaluation.

The answer to the second question is however quite well known today. In a recent survey by Mentor Grapics [McC05] on adoption of new EDA Methodologies, 27% quote "having to learn a new language" as a reason for not adopting new methodologies. The survey only considered languages relevant to industry, therefore narrowing the scope to imperative languages. It can therefore be expected that adoptions of functional languages in hardware design will only gain importance,

- if example designs have shown that functional HDLs can provide all features current imperative HDLs do,
- if automatic design-exploration becomes a pressing need for many designs (requiring suitable formal input specification to respective tools) and
- if it is acknowledged that imperative languages do not give satisfactory results by principle.

2.4 A Review of Experimental HDLs

We try to identify language features and abstractions introduced by reviewing experimental HDLs. Two basic programming paradigms exist: imperative and functional programming. We will categorize the HDLs by the paradigm chosen to implement the respective compiler. Usually this also dictates the paradigm chosen by the HDL itself.

Some of the projects presented have been abandoned while others have a long history. This list is thought to provide an overview on existing projects and to enable identification of main objectives of the languages mentioned rather than to imply any suitability for actual hardware design.

2.4.1 Functional HDLs

Hydra

Hydra is a functional HDL embedded in Haskell [O'D02]. It uses mathematical functions to model circuits and provides separate semantics for all basic constructs to enable simulation, netlist generation and timing analysis. Using distinct signal types for each task (boolean for simulation, wire for implementation, path depth table for timing analysis) allows the same circuit description to use the correct semantics based on type propagation. While all HDLs attempt to exploit parallelism when implementing a circuit, Hydra also provides mean to take advatage of the implicit parallelism of circuit descriptions for simulation, allowing distributed simulation on suitable platforms.

HML

HML is a HDL based on the functional programming language SML (Standard ML) [LL00; Li95]. HML models compile into VHDL. Emphasis of HML is on exploiting SML's advanced type system to allow more succinct models. Specifically the type inference technique enables more generic models and automatic generation of interfaces.

Lava

Lava is a functional HDL implemented as a collection of modules (embedded) in the functional programming language Haskell [Bje98]. Lava builds upon earlier work of the group on μFP [She84] and Ruby [SR93]. It exploits language features of Haskell like monads, type classes, polymorphism and higher-order functions for circuit design.

The motivation of Lava lies in the lack of suitable features in current HDL to support tasks of increasing importance in circuit design like verification. Lava was built to "be able to describe hardware at different levels of abstraction, and to analyse circuit descriptions in many different ways"[Bje98]. The tasks specifically supported by Lava are simulation, verification and implementation.

For Verification, Lava can generate input suitable for different theorem provers, while for implementation it can generate VHDL output. Details on using Lava for design of FPGAs is given in [SS].

The major aspect Lava is addressing is the concise exploitation of language features for hardware design and circuit verification.

Hawk

Hawk is a microarchitectural design language embedded in Haskell [LLC99]. Its aim is to support succinct modelling and advanced verification of current microprocessor designs. In many respects, Hawk is similar to Lava, but on a different level of abstraction. Like Lava, it exploits type classes and monads, in addition it uses lazy lists and allows signal observation via `unsafePerformIO`.

The specific aspects addressed by Lava is support of the microarchitectural abstraction level and verification.

Verischemelog

Verischemelog is a functional HDL embedded in Scheme, compiling into Verilog [JB99]. It was motivated by the observation that many hardware designers using Verilog, turned to scripting languages to generate parameterized models when exceeding Verilog's built-in capabilities. It relies on list-based syntax to provide the required flexibility. Verischemelog can be seen as an intelligent Verilog macro language embedded in Scheme. Interfaces are checked automatically for consistency. Scheme provides an interactive working environment which can also used with Verischemelog, providing direct feedback The language has been designed with extendability in mind, allowing convenient access to the language internals.

The major aspects adressed by Verischemelog are parameterizable models, interactive programming and easy language modification.

Gropius

Gropius is a functional HDL for formal synthesis [EB99]. Its guiding principles are expressiveness, unambiguous semantics and minimum size. Gropius is divided into sublanguages, each corresponding to a specific abstraction level (gate level, RT level, algorithmic level, system level). Models at all levels of abstraction are synthesizeable. Gropius was developed to serve experiments in formal synthesis. Formal synthesis describes a synthesis process where the implementation is derived from the specification by applying elementary mathematical rules within a theorem prover. As a result the implementation is guaranteed to be correct [Eis99; Kum96].

Confluence

Confluence is a functional HDL invented by Tom Hawkins in 2003 [Haw05; Max03]. It aims at improving expressiveness of models to reduce overall line count. Confluence is a compiler implemented in Objective Caml. It consists of two applications. The compiler converting a Confluence model in a FNF-model (Free Netlist Format) and a set of converters to generate C, VHDL or Verilog models from the FNF description.

While Confluence was originally a commercial product by Launchbird Inc., it was released as open-source in 2005. The main limitation of Confluence was its monolithic compiler application, discouraging modifications to the original code by others.

The aspect addressed by Confluence was succinct model description, enabled by taking advantage of higher-order functions and lists allowing writing of very generic models.

HDCaml

HDCaml is a functional language for generic hardware description at RTL [Haw06]. It is embedded in the functional Objective Caml (OCaml) programming language. Implemented as a functional library, it inherits expressiveness of functional languages and makes it available to description of hardware designs. Compilation generates executables which output models of the described hardware in different formats. HDCaml is an open-source project. Karl Flicker has analyzed and extended HDCaml to improve its robustness [Fli07].

HDCaml is the direct successor of Confluence. Beneath the succinct model description inherited from Confluence, it especially addresses the issue of extending the code of the implementation which is helped by implementing it as a functional library (rather than a monolithic compiler, as was the case with Confluence).

HDCaml has been chosen as the host language for the case studies presented in this work. Further details on its internal organization can be found in chapter 5.1 and appendix A.

Jazz

Jazz [Vui] is a functional HDL for design of large and complex digital synchronous arithmetic circuits. It is based on the theory of circuit representation in [Vui94a]. Jazz derives its type system from ML-sub, supports object-oriented design and type inference. The current environment comprises a compiler, simulators, and code generators for the Pamette

Xilinx-based board. The compiler and supporting tools are implemented using Java and the respective source code is available to the public.

Jazz was used for evaluation of the hough transform hardware implementation, enabling the first real-time transition-radiation tracker at CERN [Vui94b].

MHDL

MHDL (microwave and millimeter wave integrated circuit hardware description language) is a functional HDL for design of high-frequency circuits [BD93]. It constitutes a special case amongst all other presented HDLs as it is not thought for design of digital circuits. It is however most interesting from a language implementation point of view and is therefore included. MHDL is based on Haskell and exploits very efficiently its features. Advanced topics are discussed in [Bar95].

2.4.2 Imperative HDLs

In contrast to pure functional languages, imperative ones allow multiple variable assignments. This can make analysis of data flows very difficult, but allows natural description of state machines (relying on change of their state variables). To ease analysis complexity, many imperative HDLs aiming at data flow centric applications restrict multiple assignment, resulting in single-assignment languages (like Sassy).

MyHDL

MyHDL is a Python-based hardware description and verification language [Dec07; Goe06a]. It allows generation of Verilog code from a MyHDL specification. MyHDL models concurrency using Python generators (resumable functions). The hardware abstraction provided is register-transfer level. Testbenches can however take advantage of the full Python feature set (including especially its extensive high-level class libraries).

The specific aspect adressed by JHDL is verification, especially the extensive support for writing of efficient testbenches.

JHDL

JHDL (Just-Another Hardware Description Language) is a structural HDL developed by the Configurable Computing Laboratory at Brigham Young University, presented originally in 1998 [BH98].

It is implemented as a tool-set and class library on top of the Java programming language. JHDL is a research vehicle to explore functionality of FPGA design tools. Hardware abstraction provided by JHDL is register-transfer level. JHDL targets specifically reconfigurable platforms (FPGAs) and allows modification of the design in time. A design entity encapsulated as an JHDL-object can be moved dynamically between hardware and host-PC, allowing matching of system capabilities with current application requirements.

The JHDL tool-set features an integrated development environment (IDE) and a compiler generating EDIF output from JHDL specifications for Xilinx FPGAs. A unique feature of the JHDL IDE is the transparent presentation of design simulation (in software) and design execution (in hardware).

The specific aspect adressed by JHDL is support of reconfigurable systems, considering functional modules in hardware a movable resource.

SA-C

SA-C (Single Assignment C, also called Sassy) is a HDL based on the C programming language for design of signal and image-processing applications [HDB99]. SA-C is a *single assignment language*, implying that the value of a variable can only be set once. SA-C is a subset of C, specifically omitting pointers, multiple variable assignments and recursion. It extends however the functionality of C with multidimensional dynamic arrays, parallel looping mechanisms, variable bit-precision data types and fixed-point data types.

The user can guide generation of circuits via pragmas. A basic template mechanism for providing data-type generic functionality is provided via the preprocessor.

Like JHDL, the toolset of SA-C specifically aims at support of reconfigurable systems.

RTL++

RTL++ is an imperative HDL for describing circuits at register-transfer level, improving upon current languages by supporting *pipelined operations* (operations stretching more than one clock cycle) [ZG05]. While RTL++ has been introduced as a language by Zhao and Gajski, they state that their intention is only demonstration of features and that they "have not found an suitable language allowing [..] to define the RTL semantics we would like to propose" [ZG05, p.549]. A case study using SystemC and extending it by the central RTL++ construct of *pipelined register variables* shows a reduced line count (40%) and a more adaptable model.

The central aspect addressed by RTL++ is the efficient modelling of pipelined operations and unambiguous synthesis thereof.

Occam for Hardware

Occam is a concurrent programming language implementing the concept of communicating sequential processes [Wex89]. It was originally designed by INMOS Inc. in 1983 as the programming language for their *Transputer* computer system. Today implementations of Occam for many computer platforms exist.

While Occam is not a HDL in itself it serves well many native tasks of HDLs as its features allow for efficient analysis and hardware synthesis. This is due to the fine-grained parallelism typical to Occam, which corresponds well to the basic building blocks of low-level circuit design like distributed memory (registers), parallel execution (multiple circuit instances) and direct interprocess communication (wires). Experiments using Occam to describe functionality which can be verified in software (executable specification) and successfully mapped into hardware have shown that Occam (or other parallel programming languages) might serve well as future HDLs [PL91; PC00a].

Chapter 3

Digital Hardware Design in High-Energy Physics (HEP)

Programmable logic has become extremely popular in High-Energy Physics (HEP) experiments. Consequently, Hardware Description Languages (HDLs) have become more and more important to respective designs.

We will outline requirements which the HEP community shares with other users of programmable logic and the ones distinguishing it from other applications. We will present a number of contemporary HEP designs, using FPGAs to implement digital signal processing algorithms. Subsequently we will point out specific shortcomings in the design-flow and will discuss possible improvements.

3.1 Overview

HEP experiments are usually large projects, spanning several years, if not decades, and requiring a large number of engineers and scientist to build them. Therefore any decision taken, bears the risk of being outdated at the time it is actually implemented. Hence it is one of the aims of HEP experiments (and all electronic designs therein) to retain as much flexibility as is technically and financially feasible.

Programmable Logic allows freezing parts of a design only very late in the design-cycle. Therefore it has enabled more flexible designs since its appearance in the 1980s. In early designs, programmable logic has been used mainly as *glue-logic*, i.e. to implement interconnects between non-matching chip-interfaces.

HEP experiments tend to generate huge amounts of data (LHC beauty (LHCb) for example will generate about 3200GB of raw data per second). Processing this data both fast and accurate is a challenge, vital to any experiment's success. Therefore fast computing facilities, both on- and off-line, are an important enabling factor to HEP experiments. The experiment's subsystem responsible for collecting and filtering of data is often called the *Data Acquisition System (DAQ)*.

In the late 1990s, capacity of logic resources available had grown to such an extent that Digital Signal Processing (DSPing) became actually feasible to be implemented in programmable logic. Designs implementing DSPing on programmable logic combine the flexibility of software with the speed advantage of custom integrated circuits. The tech-

nology allowed for the first time keeping fast signal processing flexible till very late in the design process. There is however an extra effort required due to the dominating hardware-centric design methodologies.

The trend in HEP toward more digital designs and usage of related computing architectures (GPP, DSP, FPGA) has been documented in a survey by Angoletta [Ang03].

Today, DSPing on programmable logic provides a viable design alternative to

- Application-Specific Integrated Circuit (ASIC) which are fast, but are very expensive to design and have a long lead-time and
- special-purpose processors (mainly Digital Signal Processors (DSPs)), which are programmable, but lack fine-grained parallelism, require higher clock frequencies and do not scale well for many applications.

Apart from availability of Field-Programmable Gate Arrays (FPGAs) providing sufficient logic resources to implement required DSPing functionality, HDLs are the key enabling factor to DSPing designs using programmable logic (see section 3.2 below for a discussion). The VHDL and Verilog HDLs are the dominant tools for design entry in programmable logic design today. DSPing designs push however the limits of current HDLs and Register-Transfer Level (RTL)-centric design flows, rendering the design effort for advanced DSPing implementations important to inhibitive.

In the following, we will elaborate on the specific requirements in HEP, related design flow deficiencies and possible solutions.

3.2 HEP Digital Design Methodology

The life cycle of a high-energy physics experiment has three very distinct phases: design, implementation and operation.

During **design**, a feasible compromise has to be found between concurring design requirements. HEP experiments usually comprise many systems of which digital logic is only a fraction. High-level models emulating potentially hardware-based DSPing functionality will be used to evaluate the overall system's performance. HDLs might be used to implement proof-of-concept prototypes very late in the design phase.

During **construction**, the specifications of the experiment give a solid framework for refinement and implementation. Hardware designers are typically free to explore design variants within the limits of the original specifications. However, specifications might not always be given in hardware-terms, complicating proper design-space definition.

During **operation**, new knowledge is gathered constantly by analyzing data and relating it with operating conditions and particle physics model's predictions. New insights allow for refinement of models. Deeper understanding will allow tuning of respective parts of the experiment to achieve improved measurements.

Bearing the aforementioned facts in mind, algorithms performed in programmable logic (expressed using HDLs) are of very high interest because programmable logic is often the stage nearest to the detector, where modifications are feasible during operation (i.e. they do not require replacement of hardware but can be performed by reconfiguretion). Algorithms performed in hardware need however to be designed with very much care,

because operations found early in the signal chain (implemented in programmable logic) are usually lossy, i.e. they define what data will be available off line for further investigations. Hence unsatisfactory performance of algorithms in programmable can not be compensated for at later stages of data processing. (In contrast to off-line processing, where stored data can be processed multiple times, using different algorithms.)

Any programmable logic device needs a configuration bitstream, which actually implements the required functionality when sent to the FPGA. Generation of such a configuration bitstream is the task of a synthesizer. The industry-standard input to synthesizers are circuit descriptions in either the VHDL or Verilog HDL. The typical level of hardware abstraction supported for logic synthesis is register-transfer level (RTL).

Describing glue-logic at RTL is easy, as it matches the problem's native level of description. Modeling signal processing algorithms at RTL is however neither easy nor intuitive, as descriptions of much higher abstraction are typically employed for algorithm design and evaluation (e.g. mathematical notation). Generation of configurations for programmable logic devices requires however a circuit description in a HDL (RTL being the industry standard layer of abstraction). **Strategies for mapping algorithms into HDLs** (e.g. mathematical notation into RTL) are therefore of vital interest to hardware designs in HEP. Mapping models of higher abstraction than RTL into hardware is often called *higher-level synthesis*.

Designing algorithms for high-energy physics experiments (especially in data acquisition) often requires trade offs between many factors, some of them from very distinct areas (particle physics, solid state physics, Monte-Carlo simulations, analogue electronics, digital electronics, ..). Optimization of designs therefore means relating a design with results from simulations spanning a wide range of domains and potentially very different simulation environments. Hence the quality of a design is directly related to the number of relevant simulation environments for which **simulation models** can be provided. Such models can either be generated manually, or derived automatically. Simulation environments to be served could for example include numerical analysis systems like Root for statistical performance analysis, as well as custom-built C++ frameworks like Gaudi for physics simulation or tools for modelling of network traffic.

The more models circulate for a given design, the more important it is to guarantee coherence of models. Manual construction of models with identical functionality is extremely error-prone. It becomes also very laborious as modifications to one model have to be propagated manually to other models. The more models are in use and the more likely modifications to the hardware model are, the more emphasis should be put on *automatic* generation of simulation models.

Automatic generation of simulation models for a diverse range of simulation environments is therefore vital to evaluation of hardware designs in HEP.

In the former paragraph coherence of models was demanded. This is a major issue with manual model construction. But even if models are derived automatically, there should be tools to cross-check that models which are expected to be identical actually are identical. This can be achieved by formal proof of identity or by comparing output stimulated by an adequate stimulus. The larger a design and the more frequent changes to it, the more an automated test environment is necessary. As HEP design's complexity is growing and changes on short notice have been identified as an important attribute of HEP designs,

verification (comparing implementation against their specification, crosschecking derived models) is a major issue in HEP design.

HEP experiments usually require huge amounts of data to be processed in fixed time. This leads to high-throughput solutions, often only achievable using programmable logic. The complex data structure (multiple sub detectors) and patterns (particle trajectories) require advanced algorithms to separate physics events from noise.

If these algorithms need to change, designers are in need of circuit models which can be modified to follow suggested or required changes. This requires a much more responsive design process than in conventional design flows. There are two specific situations in HEP experiments causing substantial changes to circuit models:

- The design of hardware for HEP experiments often relies on extensive assumptions and simulations. If a model is flexible enough to follow shifting needs in the experiment's **initial design phase**, it can give substantial feedback on costs and benefits of different design variations. If however the model is too static to allow efficient evaluation of design variations, it can not contribute to the decision process and the final layout of the experiment might be suboptimal from a hardware design point of view.

- Requirements to an algorithm might change substantially **during operation**. While the initial design is usually embracing a more conservative perspective, researchers try to refocus the detectors, once correct operation is confirmed and operation experience increases. New discoveries or not matched predictions can trigger a major shift compared to the original intention.

  Especially new discoveries (for which HEP experiments are built, after all) can occur multiple times during the lifetime of an experiment.

The ability and speed of a circuit model to respond to *changes in the mathematical algorithm* it implements considerably defines the responsiveness of a HEP experiment to new knowledge.

How can a model increase responsiveness to unpredictable changes? In using a suitable notation, expressing a design in generic terms where changes are most probable. Typical HDLs often require explicit specification of bitwidths. If a signal is expected to change its bitwidth frequently, a more suitable expression would be to keep the bitwidth generic. If the signal represents a fixed-point number, it would be wise to use a suitable data type, defined by precision and range.

Providing matching semantics for a given application domain is known as **domain specificity** in the design of domain-specific languages. Therefore what is required to provide flexible circuit models in terms of HEP design parameters are HDLs with a high domain specificity with regard to HEP and the respective algorithms implemented.

The level of abstraction provided by current HDLs and related tools does not allow a succinct description in terms of relevant design parameters. Small changes to an algorithm from a physicist's point of view can potentially trigger a major change in the most abstract HDL model available.

3.3 HEP-specific Electronics Design Automation Requirements

In section 3.2 we have identified four factors prominent to DSP-related designs in high-energy physics (mapping algorithms into HDLs, providing custom simulation models, verification, domain specificity). In the following, we will contrast these factors with other application domains and with current state of the art in Electronics Design Automation (EDA). We will show which problems HEP designs share with other application domains and which ones are specific to HEP.

Mapping algorithms into HDLs is a common goal of all users of programmable logic for DSPing. There are many attempts to provide a consistent design flow from more abstract algorithmic descriptions to HDLs. Most prominent are currently C-based approaches which extract circuit descriptions from untimed C-code [Men07; Cel04a; GDG04; Syn07]. No generally accepted solution to this problem have evolved yet however.

It is generally understood, that verification of designs (comparing an actual implementation with the original specification) is one of the most challenging task in circuit design. Providing support for **verification** is therefore a common goal to all users of programmable logic. Recent developments include availability of assertion-based verification through integration of the *Property Specification Language* (PSL) in most digital electronics design tools [FMW05] and adoption of PSL as an IEEE standard [Psl].

Today, the use of PSL is often limited to adding assertions or properties to an already existing (and functional) RTL model. This results in redundant specification (the RTL design intention is reproduced in the PSL commands). The PROSYD project (`http://www.prosyd.org/`), sponsored by the European Union in 2004-2006 aims at providing a consistent tool flow for property-based system design. Property-based system design means using a set of higher-level properties as the golden reference model, deriving hardware implementation, tests and simulation models from it.

Providing models to different simulation environments is a requirement more special to high-energy physics. This is a consequence of the ratio between the size of the overall simulated system and the DSPing functionality in it. This ratio is especially high in HEP i.e. DSPing in programmable logic tends to be only a small part of the overall system. Most data analysis tools in HEP are custom-built or highly customized standard software packages. Clearly, support for these HEP-specific tools is lacking in tools aiming at general hardware design.

While some tools allow generation of executable models from HDL models (mostly expressed in C), no commercial tool we are aware of allows direct access to the code generation process. Therefore, adapting executable models to the requirements of a specific simulation environment is a laborious, in the case of closed source unfeasible, task.

Domain specificity of models: In most digital systems, programmable logic helps to implement a well-specified functionality (top-down design approach). Typically, the more flexibility is required from a solution, the more likely it is to be implemented in software (where changes are easier and less costly). While this design trade-off certainly applies to HEP designs as well, the importance of specific factors might deviate considerably when compared to their respective importance in other application domains. Potentially, this leads to more complex algorithms being implemented in programmable logic. This in turn

leads to HEP hardware designs spearheading development and being ahead of required tool support.

The requirement to adapt to new knowledge at short notice is prominent in HEP designs. The combination of complex algorithms and the need to follow high-level changes to the algorithm requires a domain specificity of circuit models unmatched by other application domains.

3.4 Review of Selected Designs

As outlined above, FPGAs are a viable platform for implementation of DSPing algorithms. This section will give examples of existing designs in HEP using FPGAs for *advanced DSPing*. We define "advanced DSPing" by the effort required to implement the DSPing-part of the design, compared to the general design effort for the respective FPGA. If either the estimated design effort exceeds about 30%, or if the DSPing-part is foreseen to change over the system's lifetime, we will consider the system.

Due to the sheer amount of electronic designs in HEP and the widespread literature, such a list can only represent an arbitrary sample. The sample given is certainly biased toward systems developed at CERN and being related to the large-hadron collider (LHC). It should however be generic enough to demonstrate scale of current systems, typical design approaches and identified deficiencies thereof.

Each system will be characterised by the following data:

- Brand and type of FPGAs used
- FPGAs per board

  We will only mention FPGAs related to DSPing functionality. Often further FPGAs can be found on-board, implementing interfaces or other low-level functionality.
- Identical boards in parallel use
- Logic resources used, clock frequency
- typical input/output data rate

  One has to distinguish between the maximum data rate defined by the interface and the actual data rate used by the application. Especially when low latency is an issue, systems tend to feature very fast interfaces, but use only a fraction of the available bandwidth. To estimate available bandwidth for future upgrades, it is of interest to know both. Where available, we will therefore give both figures.
- primary task
- chosen design flow

Generally we can distinguish HEP systems for DAQ and for (beam) control.

In *DAQ*, the main task is data reduction to handle the excessive amount of data collected by sub-detectors and to extract events of interest (physics events). Extensive input bandwidth, high throughput, on-board multi-channel processing and multiple boards

working in parallel are characteristic for this application. DAQ systems usually consist of a number of cascaded trigger levels. The lowest level receives unfiltered data and tries to come to a decision in very little time by considering data only from certain regions of interest. Data identified as belonging to a physics event is passed on to the next higher level, all other data is rejected.

In *Beam Control*, the signal processing is part of a control loop. Low latency is key. Systems are usually much smaller in terms of bandwidth and boards, but algorithms tend to be more advanced and tuned more toward fast results (low latency) than for parallel processing.

LHCb vertex locator zero suppression

The LHCb VErtex LOcator (VELO) provides precise track coordinates close to the interaction region of LHCb, one of the four experiments installed at the LHC. The VELO is based on silicon strip sensors. Two semicircular sensors with 2048 azimuthal or radial strips respectively form a module, of which 50 are positioned along the beam axis (2 x 2048 x 50 = 204800 strips in total).

All strips are read out in parallel with the LHC collision rate of 40.08MHz and stored in analogue memory. From there data is transmitted via twisted-pair cables to the LHCb DAQ interface board (TELL1) upon reception of a trigger signal (not exceeding 1.1MHz in average).

At each event, only a low percentage of channels will carry interesting information. Data volume of events can therefore be reduced significantly by zero suppression (removing noisy channels void of physics information).

The TELL1 digitizes incoming data and performs Digital Signal Processing (DSPing), specifically pedestal subtraction, common mode correction, zero suppression and data formatting. For reasons of performance and flexibility, all DSPing is implemented in FPGAs.

The TELL1 is the common LHCb DAQ interface board [Hae06]. It is not only used by VELO, but by most sub detectors of LHCb. There are a total of almost 300 TELL1 boards installed at the LHCb cavern. The input stage is realized by mezzanine boards for either analogue (twisted pair) or digital (12-way optical fibers) data. The analogue version as used by VELO provides a total user bandwidth of 25.6Gbps per TELL1. The incoming, digitized data stream is processed in parallel by four preprocessing (PP)-FPGAs (Altera Stratix 1S25), each accompanied by 96MB local DDR SDRAM. A fifth FPGA (Altera Stratix 1S25) collects the processed data from the PP-FPGAS and formats it for transmission to a dedicated PC-farm via GigabitEthernet.

The TELL1 comes with a VHDL framework, providing basic functionality, which can be extended by the respective subdetectors to adapt signal processing to their needs. Design entry is based on VHDL and schematic capture using MentorGraphics HDL Designer.

The TELL1 framework provides also test and monitoring features like in-system data generators and access to on-board RAM and register. There is as well a full suite of C++-models for the basic algorithms provided.

To ease keeping C++ simulation models synchronized with the hardware implementation, an approach deriving both models from one common source has been investigated [MS06] (see also chapter 5.2). The effort was however started too late and will not be used in the system due to be ready for the first LHC beam end of 2007.

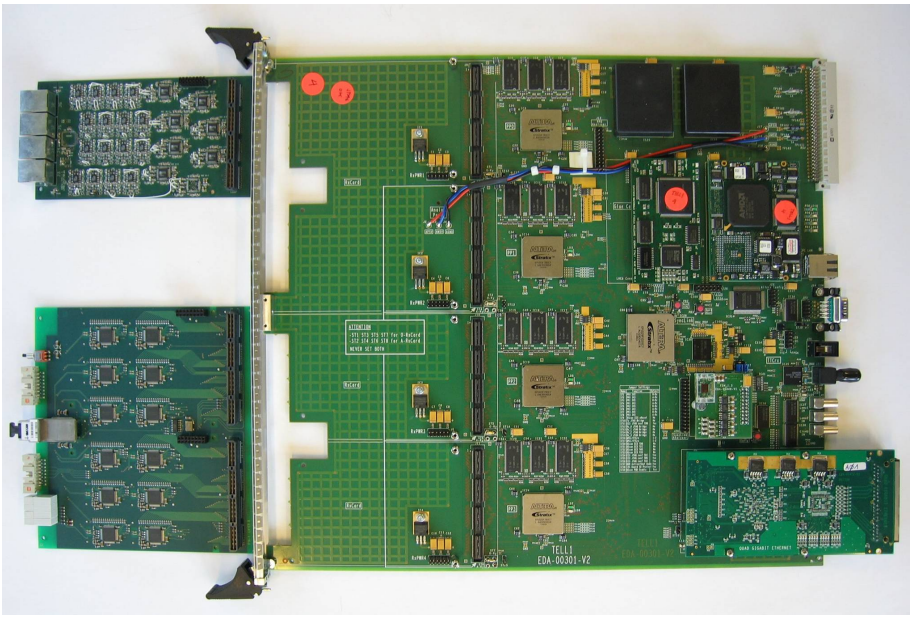


Figure 3.1: TELL1 with optical and analogue receiver cards to the left (courtesy of Guido Haefeli)

LHC Beam Loss Monitoring processing module

The Large Hadron Collider (LHC) beam loss monitor [Hol05] (BLM) is an important part of the LHC machine protection and quench prevention. Each turn, beam energy is measured by 4000 ionization chambers. Data is collected by 650 data acquisition cards located in the LHC tunnel. From the tunnel, data is transmitted via optical links to the surface where turn-by-turn energy and variations thereof are calculated in real-time. The combination of high- and low-precision measurements, calculation of total beam energy, loss patterns and checking against given tolerance curves is performed in programmable logic [Zam06] on the BLM processing module.

The main motivation for putting this functionality into FPGAs as quoted in [Hol05] was: meeting demanding timing requirements, flexibility with regard to future upgrades or changing system specifications.

The BLM processing module is a VME card named DAB64x [Deh06] (see figure 3.2). The distributed processing system consists of about 350 boards, each carrying one Altera Stratix EP1S40 FPGA, 1MB Flash and 3x2MB SRAM. The current design requires almost all logic resources available (97% logic element usage).

Each DAB64x board carries a BLM mezzanine card featuring four optical gigabit link receivers providing low-latency communication with the cavern. While the theoretical maximum input bandwidth is 4 x 800Mbps = 3,125Gbps, only a small fraction (320bit x 25kHz = 8Mbps) is actually used. The BLM is a security control system, intervening only if the beam is lost, therefore no continuous output data stream is necessary and the output bandwidth is negligible (a VME PC reads data with 1Hz for monitoring).

The chosen design methodology is mostly VHDL at lower level and schematic capture at higher level. One design issue specifically mentioned in [Zam06] and not supported by

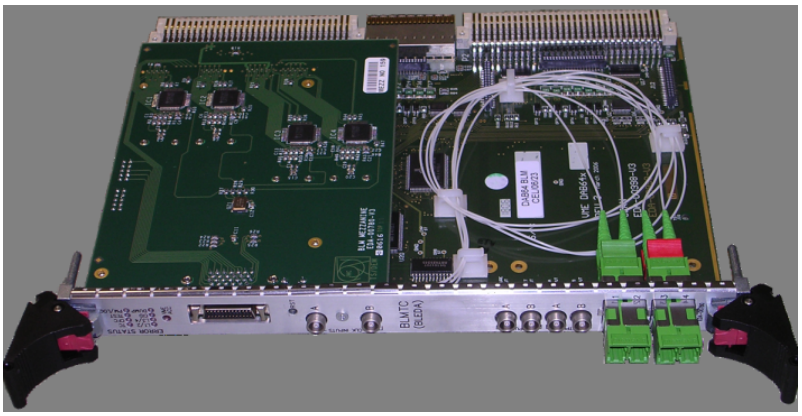


Figure 3.2: Beam Loss Monitor Processing Module (DAB64x) with mezzanine board [Deh06].

tools is the optimization of memory resource usage (requiring resource sharing between different building blocks).

The FPGA configuration is expected to change rather frequently and major updates have already been realised.

LEIR low-level radio frequency system

The CERN Low Energy Ion Ring (LEIR) low-level radio frequency (LLRF) system is a fully digital beam control system. Commissioning took place in 2006 with prior tests performed in the PS Booster accelerator using a scaled-down system.

The very task of a LLRF system is control of the cavities RF voltage to provide a stable beam and requested acceleration. Due to the very low latency imminent to this control task, LLRF systems were typically implemented in analogue electronics until recently. Today, advances in digital technology allow fully digital LLRF systems [Ang06]. The most prominent advantages of digital LLRF systems are: reconfigurability without hardware intervention, multi-user operation (parameters on per-user basis), reproducibility, absence of drifts caused by analogue devices and the ease with which built-in diagnostics can be implemented.

The LEIR LLRF system achieves the required speed and flexibility by using both FPGAs and DSPs. The FPGAs act as fast pre- or co-processors, providing processed data to real-time loops running on the DSPs.

The system used for the 2004 test [Ang05a] comprises one DSP carrier board carrying three daughter cards, namely:

1. DDC: a four-channels, two-sites Digital Down Converter, performing digitisation, low-pass filtering and decimation of analogue signals. Originally a commercial chip (Intersil ISL5216 DDC chip) was foreseen to provide the required functionality. Due to a phase discontinuity discovered in the prototype design, this approach was dropped and all relevant functionality was implemented in a FPGA.

2. MDDS: a one-site Master Direct Digital Synthesiser, generating, under DSP control, a high-frequency analogue clock signal from a 100-MHz clock reference;

3. SDDS: a one-channel, one site Slave Direct Digital Synthesiser, generating, under DSP control, an analogue signal from the MDDS-generated clock reference. This board features an Altera Flex FPGA. Two look-up tables allow fast calculation of sine- and cosine functions to convert I/Q data.

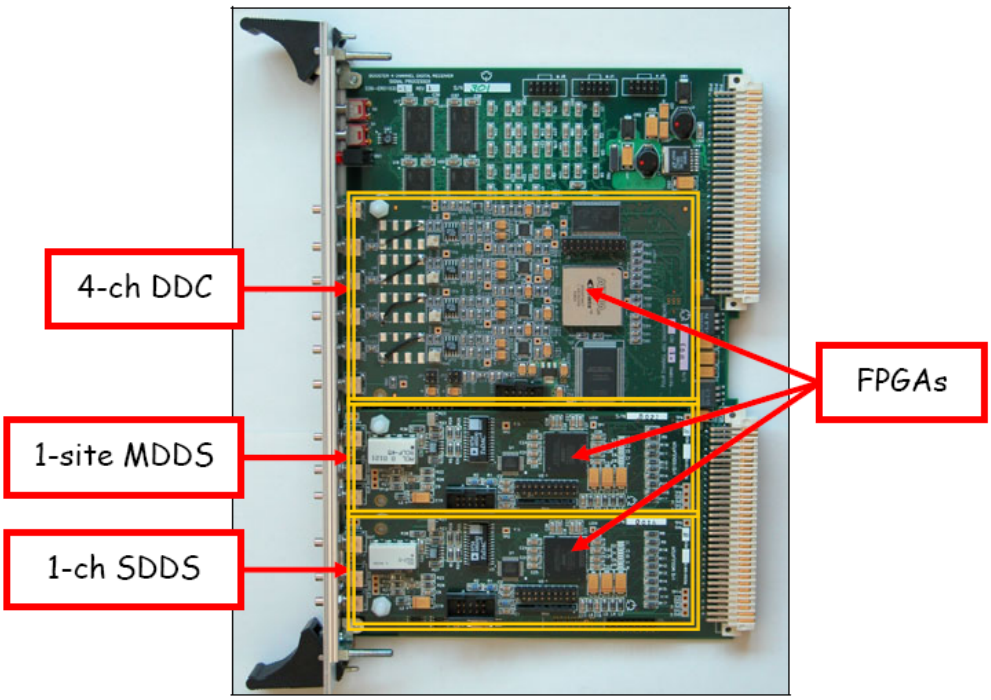


Figure 3.3: LEIR DLLRF DSP board equipped with daughter boards (picture taken from [Ang05b])

The carrier board is a 6U VME board. The used DSP is an Analog Devices ADSP-21160. All daughter cards and the carrier board itself feature one FPGA each.

The system has been developed in cooperation with Brookhaven National Laboratory, Upton, USA where it is used for control of the AGS Booster accelerator. The system is foreseen to be employed to different accelerators at CERN (LEIR, PSB, PS and AD), resulting in more uniform hardware and improved focus of further development.

An important problem requiring further attention is the limited availability of low-latency communication links between DSPs (limiting available computing power for low-latency applications).

CMS Endcap Preshower Data Concentrator Card

The Compact Muon Solenoid (CMS) is one of the four LHC experiments. CMS' Endcap Preshower (ES) sub-detector comprises 4288 silicon sensors, each containing 32 strips. The signals from each strip are amplified, shaped, sampled every 25ns and stored in an analogue memory. On reception of a trigger (max. trigger frequency 100kHz), three

consecutive samples per strip are digitized, formatted and sent via optical links to the ES data concentrator cards (ES-DCC) located in the counting room.

The task of the ES-DCC is to receive data from up to 36 optical links, to verify data integrity and to perform on-line data reduction [Bar06; MSV06]. The bandwidth required is typically reduced by a factor of 20. The steps performed to achieve data reduction are: pedestal removal, channel calibration, common mode rejection and zero suppression.

The ES system contains approximately 40 ES-DCCs. The ES-DCC is a 9U VME board, equipped with 7 FPGAs:

- 3 *Reduction FPGAs* performing de-serialization of 12 incoming data streams each and subsequent data reduction. The functionality of all three FPGAs is identical, providing sufficient resources for parallel processing of data from 3x12 optical fibers.
- 3 *Spy FPGAs* to provide on-line monitoring services via the VME bus. Each FPGA has direct access to 3x18MB local static RAM. The FPGAs are less powerful than the reduction FPGAs (Altera Cyclone)
- 1 *Merger FPGA* collecting resulting data streams from the three reduction FPGAs and formatting of them.

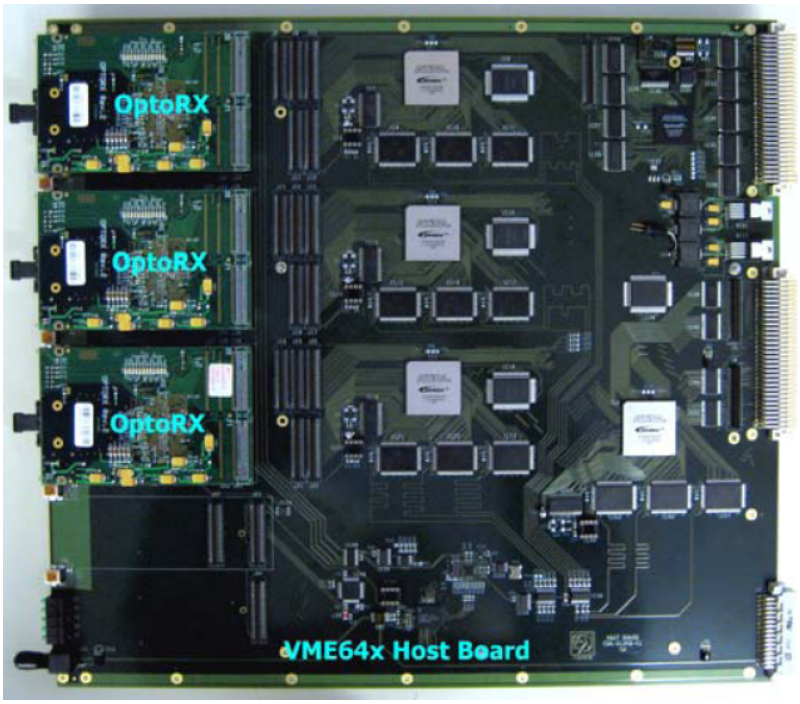


Figure 3.4: CMS Endcap Preshower data concentrator card (ES-DCC), equipped with three OptoRx12 receiver/processing modules (picture taken from [Bar06])

The reduction FPGAs are actually mounted on a small carrier board each, the OptoRx-12, providing a modular and flexible system [VR06]. The same module is used (and was developed in cooperation with) the TOTEM experiment. The OptoRx-12 can be equipped with either of two devices from the Altera Stratix GX family: 1SGX25 or 1SGX40. The

decisive factor for choosing the type of FPGAs used was the extensive number of distributed on-chip memories (1SGX25: 224 512bit blocks, 138 4kbit blocks, 2 512kbit blocks) allowing implementation of multi-channel processing [VR06].

The theoretical input bandwidth of the ES-DCC is 36 x 800Mbps = 28.1Gbps, the actually used data rate at maximum trigger frequency is 600byte x 36fibers x 100kHz = 16.8Gbps. The processed data is sent to the CMS data acquisition system via SLink, providing a bandwidth of 200MBps.

ALICE HLT pattern recognition

ALICE is one of the four LHC experiments. The central detector of ALICE is the Time Projection Chamber (TPC). It consist of about 600 000 detector channels, resulting in a worst-case data rate of 75MB per event. The given bandwidth to permanent storage is 1.25GB/s, enabling storage of about 20 events/s. The High Level Trigger (HLT), a massive-parallel computing system, has been designed to perform

- real-time pattern recognition for on-line event selection and
- real-time data compression of selected events to increase data storage efficiency.

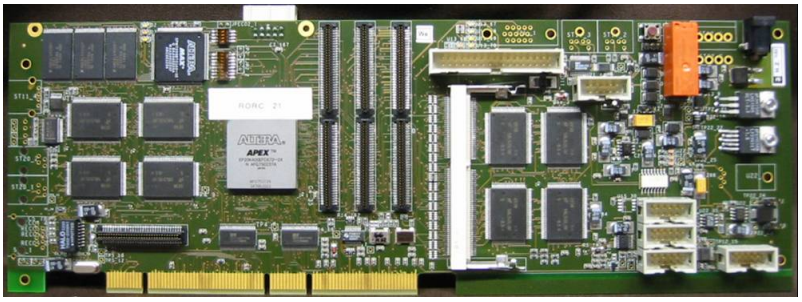


Figure 3.5: Prototype of the ALICE High-Level Trigger Read-Out Receiver Card [Gra03]

The TPC is divided into 36 sectors, consisisting of 6 subsectors each. The data from each subsector is read out and send via optical link to the reception nodes of the HLT, called HLT ReadOut Receiver Cards (HLT-RORC, see figure 3.5). Each HLT-RORC features an FPGA co-processor to support data-intensive local tasks of the HLT pattern recognition algorithms [Gra03].

ATLAS Level2 Transition Radiation Detector (TRT)

ATLAS is one of the four LHC experiments. The ATLAS Transition Radiation Detector (TRT) uses straw tubes as sensing elements. It consists of a central barrel part (52 544 straws parallel to the beam axis) and two end-caps (319 488 radial straws). The information of all straws provides a two-dimensional position measurement for all particle tracks within the acceptance of the detector. Regions of interest can be identified by searching local neighbourhoods only. For further selection, an unguided track search in the full TRT data set is necessary.

An algorithm for identification of tracks in the full data set, the TRT-LUT, is described in [Bai99]. It uses the Hough transform to reduce the search space. As the interaction point is known to be contained in each track, the transform for each position in the detector volume can be precalculated. A histogram is filled with the transformed hit data. Track detection can subsequently be reduced to finding maxima in the filled histogram.

Khomich et al. [Kho06] have evaluated achievable speed-up of a hybrid CPU/FPGA implementation in comparison to a CPU-only implementation. The algorithm was implemented in C++ and profiled. The most time consuming tasks were identified and manually reimplemented in VHDL.

For evaluation of the synthesized VHDL model the MPRACE board (Multi-Purpose Reconfigurable Accelerator/Computing Engine) developed by the University of Mannheim was used. MPRACE is a 64bit/66MHz PCI card featuring a Xilinx VirtexII FPGA.

Only the initial track finding was implemented in VHDL and synthesized for MPRACE. Thresholding, maximum finding and further tasks were executed on the CPU. The implementation required all of the VirtexII memory internal blocks (plus an external 9MB SRAM memory) and about 80% of the FPGA's logic resources.

There were two approaches pursued to allow integration of the circuit models into the existing ATLAS simulation framework:

1. Interfacing to a VHDL simulator through its programming language interface (PLI). This approach was not investigated further due to the ATLAS decision not to use custom hardware.

2. Using SystemC as modelling language. While it would have allowed easier integration into the simulation environment, the lack of SystemC synthesis support has rendered this option unattractive.

The reported speed-up is about 3 for the entire application. Higher speed-ups are expected with faster FPGAs and wider memory busses. While the project has proven the principal feasibility, ATLAS has decided not to use custom hardware in the Level2 Trigger. Therefore this hardware will not be installed in the ATLAS experiment.

Summary

The reviewed designs all implement DSPing functionality on FPGAs (this was the selection criteria).

Except for one, all designs are final designs to be integrated into HEP experiments. Therefore they serve well to document the current state of the art in advanced HEP DSPing designs and to extract respective design needs. They do reflect real engineering experience rather than projected technology usage.

The designs' relevance to the success of the respective experiments is important, supporting the claim that DSPing on programmable logic plays a vital role in current HEP experiments. The number and size of the boards, the complexity of the algorithms to be implemented and the amount of firmware to be produced shows that considerable resources (money and manpower) had to be assigned to the respective projects. We can therefore conclude that a non-negligible part of HEP experiment budgets is bound by DSPing designs on programmable logic.

Design	Type of FPGA	FPGAs/ board	Boards	Resources used	f_{max}	Input bandwidth	Output bandwidth	Primary task	Design entry	Tools
				[LE]	[MHz]	[MB/s]	[MB/s]			
LHCb VELO	Altera Stratix EP1S25	5	289	80%	120	25.6Gbps	3.84Gbps / 2.34Gbps	pedestal subtraction, LCMS, zero suppression	VHDL, schematic	Mentor HDL Designer, Quartus, ModelSim
LHC BLM	Altera Stratix EP1S40	1	350	97%	40	3,125 / 8Mbps	-	Beam Energy estimation	VHDL, schematic	Quartus, ModelSim
CMS ESDCC	Altera Stratix GX 1SGx25	3 + 4	40	90%	40, 80	28.1, 16.8	1.56	pedestal subtraction, mean calculation, zero suppression	VHDL	Quartus, ModelSim
LEIR DLLRF	Altera Stratix EP1S20	3	1	50 - 90	-160	-	-	Digital Filters	VHDL	Quartus, ModelSim, Visual Elite
ATLAS LVL2 TRT	Xilinx VirtexII XC2V3000	1	1	80%	64/128	-	-	Track selection by Hough transform	VHDL	Aldec ActiveHDL, Mentor Graphics LeonardoSpectrum

Table 3.1: Characteristics of different digital signal processing designs in HEP using FPGAs

All designs use VHSIC Hardware Description Language (VHDL) for circuit specification (i.e. FPGA configuration bitstreams are generated by synthesizers taking VHDL models as input). While VHDL is not the only HDL available for circuit specification, it is clearly the language of choice in the reviewed designs. Consequently it is a valid assumption that any HDL extension needs to be compared to VHDL when investigating its potential impact on current design flows and efficiency.

If cycle-accurate models were provided, they were maintained by hand. It is acknowledged by designers (while not frequently mentioned in publications) that providing suitable simulation models and keeping them synchronized with the hardware implementation is one of the main concerns and also requires significant (often originally not foreseen) manpower. This supports the claim that automatic generation of cycle-accurate simulation models is a major need of HEP designs, not met by the current design flow.

Typical clock frequencies achieved are in the range of 80-120 MHz. This is about the common frequency range achievable with current FPGA technology without major tweaking of VHDL models. To exploit higher clock frequencies (theoretically current FPGAs can achieve up to about 250MHz), the effort of required model optimization increases disproportionate.

This fact is important in view of the LHC collision frequency of 40.08MHz. Current designs therefore allow one (40MHZ) to three (120MHz) sequential operations between collisions. Higher requirements to future LHC designs therefore can hardly be met by increasing the clock speed. At least not without either investing more time in optimizing models or changing the design methodology. The logical consequence is that designs need to further exploit either parallelism or pipelined architectures. All options show a need to considerable intervention to the original models. It is likely that many challenged hardware models will be rewritten from scratch if increased computing power is demanded. This might provide an option to experiment with new hardware design methodologies.

In all designs, resource usage of FPGAs is high ($\sim$80%). FPGA place and route tools typically fail or become very slow if FPGA resource usage is above $\sim$80%. This supports the claim that the complexity of HEP designs challenges current technology. At such high resource-usage and speed levels (see above) having direct control over implementation details at RTL becomes important to achieve working designs in case automated procedures fail. This clearly contradicts the need of succinct high-level models required to model the complex mathematical algorithms in suitable terms.

Very limited information is given on possible design trade-offs and if or how they were evaluated. Usually all design decisions have been taken using a preliminary C/C++ model. No publication reports on a feedback from the HDL design to the physics performance. All publications give the impression that the VHDL model is a static implementation. Possibly the genericity of hardware models is seen as being of little interest such that respective data is not reported in publications.

Chapter 4

An Improved HEP Hardware Design Process

In section 3.2 we have identified four key issues for design of High-Energy Physics (HEP) Digital Signal Processing (DSPing) on programmable logic (mapping algorithms into Hardware Description Languages (HDLs), verification, custom simulation models, domain specificity). We have argued in section 3.3 that the third and fourth factor are of crucial importance to HEP designs while of less interest to other application domains. Therefore they can be considered *HEP-specific Electronics Design Automation (EDA) needs*. In the following we will outline how these specific EDA needs (custom simulation models and domain specificity) could be met by modifications to

- the hardware design process,
- the way HDLs are designed,
- type and implementation of HDL features.

4.1 Custom Simulation Models

Hardware Design Process

To satisfy the need for custom simulation models, the design process should be such that models for relevant simulation environments can be provided for each stage of modelling in digital design. To guarantee coherence of models and to allow for fast turn-around times, generation of respective models needs to be automated. It is not possible to specify exactly what "relevant HEP simulation environments" are. A generic approach to support a range of simulation environments and to add support for future environments is therefore crucial.

Custom simulation models are perfectly in line with the V-model, requiring test of the specified functionality at each level of abstraction. Each test requires a suitable model to be connected with the respective test environment. Actually the lack of suitable support in current tools can be considered a poor implementation of the V-model.

From a HW/SW-Codesign point of view, the importance of custom simulation models for hardware designs is somewhat lower-ranking with regard to the central issue of

HW/SW-partitioning. HW/SW-Cosimulation however deals with the question of how to simulate models comprising functionality to be mapped into hardware and software respectively. If Cosimulation is used for evaluation of cost functions requiring specific simulation environments, the situation is comparable to the one we describe. Custom simulation models for hardware designs are therefore a subproblem of HW/SW-Codesign.

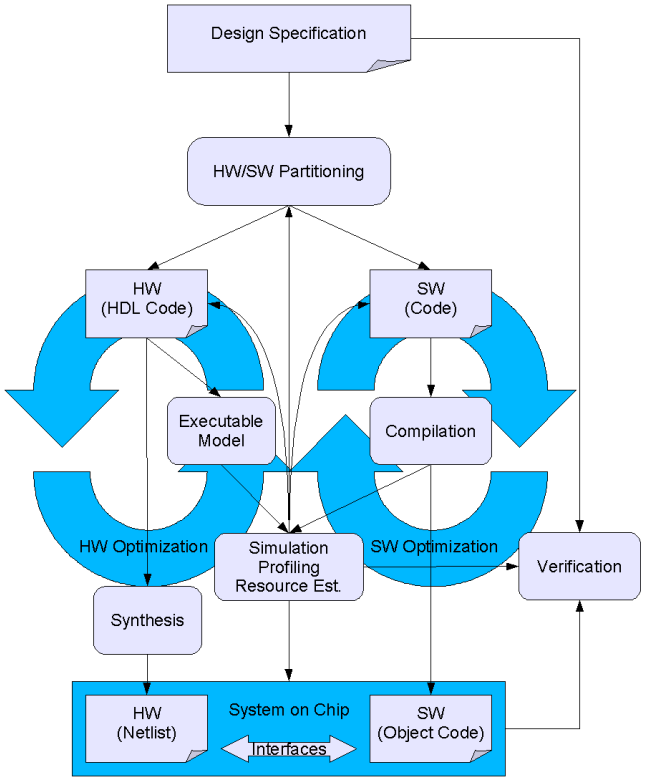


Figure 4.1: Design Flow for System-on-Chips (SoCs)

HDL Design

From a computer science point of view, generating custom simulation models from a given hardware specification is *cross-compiling* (generating executable code for targets different than the host platform). Most EDA tools targeting FPGAs are actually cross-compilers, as they usually support at least two targets (FPGAs for implementation and CPUs for simulation). Because of the complexity of current HDLs, highly specialized tools have evolved targeting either hardware (HDL Synthesizer) or software (HDL Simulator). Integrated Development Environment (IDE) only combine these tools to make them more accessible to the user.

Most current simulators read in HDL models, transform them into native code for the host platform (PC or Workstation) and feed input stimuli to these fast, executable models. Theoretically, custom simulation models could be derived by suitable changes

to the HDL simulator's code transformation module. These module's are however at the core of HDL simulator manufacturer's business models and therefore undisclosed and also highly customized to meet the needs of the specific product.

What is needed to make generation of custom simulation models feasible, are accessible transformation modules of HDL simulators. To reduce the effort required for modifications, these modules should be as simple as possible. This can be achieved by well-specified, reduced-scope HDLs. This is opposing the approach followed by VHDL which supports a wide range of domains and features very verbose syntax, making construction of tools very expensive.

What is needed therefore to allow implementation of generators for custom simulation models is an existing, accessible HDL simulator whose code transformation module is simple enough to be modified with limited effort. One could consider this an extensible HDLs simulator platform.

4.2 Domain Specificity

Design Process

The V-model suggests stepwise refinement of models. A modelling language spans a design space in which a model can be transformed. Usually the accompanying tools provide means to facilitate the respective (manual, user-guided or automatic) transformations.

Each step in the V-model comprises the relevant models at a given level of abstraction. Ideally, one model should fully specify the design at a given level of abstraction.

A domain-specific HDL increases the levels of abstraction at which the model can be used. Therefore a HDL with increased domain-specificity allows covering several steps in the V-model with the same model.

HDL Design

General-purpose programming languages usually have a long lead-time till new features are integrated into the language. This is even more true for HDLs. While in general-purpose programming languages extensive analysis of existing designs can be performed by language designers, in domain-specific languages the task is usually much more ad-hoc and carried out by users. Domain experts (users of domain specific-languages) can give qualified feedback on what features suit best the needs of their applications. They usually lack however the knowledge on how to integrate suggested features in a language. If however the user-base for a language is small and/or the commercial interest in customizing it is low, the task stays with the domain experts rather than being passed on to language designers.

Any language platform providing *means for easy extension* by users will benefit from increased feedback. This is crucial for domain-specific languages and even more important where the domain can not be clearly defined (and features therefore not foreseen in advance), as is the case in HEP.

To enable domain-specific HDLs for HEP it is therefore mandatory to provide a working general-purpose HDL which is easily *extendable*.

Two important concepts to facilitate implementation of unforeseen language features are *multi-stage processing* and *type inference*.

Extendability

The literature on domain-specific languages knows several implementation patterns for DSLs [MHS05](see also section 2.3). While DSL implementation is usually a task performed only once, some of the DSL implementation patterns are also suitable to keep the resulting language permanently open to further extensions.

Embedding refers to extending an existing language by defining new abstract data types and operators. The applicability of resulting extensions heavily depends on the user-definable operator syntax and type system provided by the host language.

Embedding is most useful if host language notation and domain notation share common properties (e.g. if both use infix notation for arithmetic operators) and if the type system suits the target domain (dynamic typing will be of little help when targeting hardware).

If embedding is not flexible enough to accomodate the domain-specific extensions, *extending the language implementation* or *using a pre-processor* allow for concepts and syntax not foreseen in the host language.

Extending the language implementation is the most powerful option but it is also known that compilers are particularly hard to extend [MHS05, p.330]. If however the compiler has been designed with extension in mind, interfacing domain-specific extensions can become easier. Kiczales and des Rivières [Kd91] have suggested the *metaobject protocol*, an object-oriented interface for specification of language extensions and transformations (used by the Common Lisp object system (CLOS)).

Pre-processors can help to add specific notation to a language and are also powerful in adding flexibility to originally static language constructs (see e.g. C++ templates). The drawback of a pre-processor is that debugging can become very difficult as the functionality lies outside the scope of the compiler.

To provide a HDL which enables addition of domain-specific features not foreseen at language design and by non-experts (i.e. domain experts rather than language design eperts) in an efficient manner, a HDL is required, featuring

- a versatile type system
- powerful user-definable operator syntax and
- defined interfaces to access the compiler implementation in case the former two options are not sufficient.

Multi-stage processing

HDLs have to describe circuits unambiguously. Either because they are used for modelling or for synthesizing of a circuit. As a hardware design is static (excluding run-time reconfigurable systems from our observations) HDLs are necessarily statically typed languages (i.e. the type of each symbol is known at compile time).

HEP requires however flexible models for evaluation of design variations. This flexibility is contradictory to the level of detail typically required for circuit descriptions and to compile-time type checking.

A good example for this dilemma is the VHDL requirement of locally static range bounds. *Locally-static* expressions are expressions whose value can be evaluated at analysis of the respective design unit[1] [IEE02, p.113], i.e there is no inter-unit dependency to be resolved. This works well if parameters are directly specified by local or global constants, but it fails if a more elaborate rule is required to evaluate the expression. One consequence is that in VHDL it is impossible for an instantiation of a function to calculate any of its own input- or output-signal's types (or parts thereof) as this would require combined evaluation of the function's declaration and body (implementation). VHDL language design trades model flexibility against implementation complexity (see [LMR94] for a discussion on implementation complexity of specific VHDL features).

Possibilities to allow more flexible models, yet assure a high level of detail and static types arise when a model is processed in multiple steps. Basic implementations of multi-step processing are macros and preprocessors. More advanced features are provided by multi-pass compilers, i.e. a compiler which processes the code (or resulting internal representation) multiple times, building each time upon the preceding result. Languages which build explicitly upon the features requiring multi-pass compilers are called *multi-stage languages* [Tah99].

If specialized languages exist, serving well a specific sub domain, it can be useful to use *generative programming* to produce output in the given language for specific targets. This also allows to rely on existing tools for the used language. VHDL for example serves very well the purpose of hardware synthesis and there is an unmatched range of high-quality tools available.

To enable flexible yet detailed models, some kind of multi-stage HDL is required, providing means for predictable design variations at different levels of detail.

To take advantage of existing tools, to blend into existing design flows and to minimize work on reimplementations of already existing solutions, generative programming to produce matching target models is required.

Type inference

> "Type inference is the process of determining the types of expressions based on the known types of some symbols that appear in them." [MA01, p.135]

Most low-level computer programming languages derive their basic data types from features of the underlying computing platform. A typical example would be the integer type in C which derives its range from the available bitwidth of the host system. This is motivated by the fact, that the operand bitwidth is hardwired and deviation from it provides no advantage in execution speed or resource usage.

In hardware design however, any single operand's implementation (including bitwidth, but also encoding, wire length, ...) can (and should) be optimized. It is therefore favorable to replace the inflexible type set of CDLs by a more flexible family of types when designing a HDL. One (negative) example is the VHDL `bit_vector` type, which can be instantiated with a specific bit width. VHDL requires however explicit, locally static specification

[1] A design unit in VHDL is either the declaration of an entity, configuration or package, or the body of an architecture or package

of the bitwidth at declaration. This leads to very static and inflexible models. It also leads to redundant information distributed all over the model's code which results in low maintainability.

Type inference is the ability of a compiler to derive automatically type information for a given symbol from a set of expressions, based on information from the types of already resolved symbols. The ML programming language provides type inference. The types of the small example `fun inc (x) = x + 1;` are resolved as follows: `1` is of type `int` in ML (the real number would be written as `2.0`). The operator `+` only allows combination of same types (int and int or float and float). Therefore `x` must be of type `int`, too. As the resulting type of an addition is specified to match the input types, the resulting type of the function `inc` needs to be `int`. ML reports the result of the type inference procedure as `val inc = fn : int -> int` (read as: inc is a function accepting an `int` as input and returning a result of type `int`).

Type inference enables a coding style, where relevant information is only given once and subsequently propagated through the model. It enforces integration of knowledge into the language or model rather than translation of knowledge into hardware terms by the designer.

Arithmetic operators for example can require the result to be of the same bit width as the operands (accepting the situation of overflow) or to be of a bit width accommodating the full range of possible results (range-guarding). If the respective bit width of operands is specified by hand, it is difficult for a designer (and impossible for a tool) to find out if a given expression actually implements the desired operation or is the result of a coding error. Type inference requires specification of the respective rule (how to calculate the resulting bit width) and allows checking by both designers and tools.

The fact that relevant type information is given in a non-redundant manner enables efficient generation of design variations. In this respect, type inference is actually the consequent extension of the well-known coding guideline to use constants for specification of values expected to change during a model's lifetime.

Type inference is therefore identified as a vital prerequisite to allow formulation of flexible, compact and maintainable HDL models.

No HDL in broad use today (VHDL, Verilog, SystemC, all C-like languages) provides type inference, which greatly restricts flexibility of models. One possible reason why type inference has not penetrated HDLs yet might be the fact that it is typically a feature of functional programming languages. Functional languages typically serve a community very distinct from the hardware design community bound more towards imperative languages. There is however no technical reason restricting implementation of type inference to functional languages [MA01, p.135].

Language Features

Domain-specific language features enable writing of models using a domain-specific notation rather than a target- (hardware-) specific one. It allows formulation of the model in terms of the relevant design parameters rather than in terms of final building blocks. This leads to more generic models, which can be modified more easily.

Extracting domain-specific notations and matching them with feasible language extension is an iterative and difficult process. It relies on domain-experts identifying shortcom-

ings of current models and on compromise between domain-specificity and general language design.

A major limitation in circuit models, implementing digital signal processing, is the fact that most HDLs lack types corresponding to *mathematical number sets*.

The often encountered combination of a given set of n bits to form a *natural number* of range $0 \ldots 2^n - 1$ results in explicit coding of the required bitwidth. This contradicts both mathematical intuition and good coding style (but has become common practice in HDL coding).

A more appropriate approach are range-based arithmetic data types. Operators could calculate the resulting range of their respective operation, given the operands' range. Such data types would allow models where range information is only given where necessary (inputs, specific intervention points). Changes to such models would be much easier and less error-prone than multiple manual bitwidth specification.

Rational numbers are important to most DSPing models, but typically lack support in HDLs. HDL models usually map rational numbers into fixed-point data types (if available, in simple bitvectors otherwise), i.e. a bit vector where a given amount of bits is used to represent the integer part and the fractional part of a signal respectively. For the two parts, the same problems as for natural numbers apply. If taken together however, the additional issue of correct binary point positioning after operations appears.

For modelling of arithmetic operations, a suitable fixed-point data type is therefore necessary. Such a data type would allow range- and precision-independent model formulation. As precision information is available, varying precision scenarios could easily be evaluated. See the following case study (section 5.3) for a more detailed explanation.

Models at register-transfer level require manual placement of registers (i.e. explicit specification of the combinational functions to be evaluated in one clock cycle). The achievable clock frequency directly depends on the longest resulting path between two registers. This in turn depends on the result of the routing algorithm which depends heavily on type and size of a design. It is common that designs, running at a given frequency when only a low percentage of a FPGA's logic resources are used, achieve a much lower frequency when the percentage of used logic resources is high. This results in manual intervention very late in the design process to reduce complexity of large combinational functions.

In most advanced designs (and especially in DSPing designs) manual specification of combinational functions to be evaluated in one clock cycle is of little interest. What is important is that complexity of combinational logic between registers is well-balanced over the whole design (to achieve maximum clock frequency) and that clock frequency tuning interventions - if necessary - do not affect arithmetic functionality.

A concept which enables such a scenario are pipelined signals, i.e. combinational functions which are allowed to require more than one clock cycle. This allows writing of models where throughput and latency can be guaranteed, but gives freedom to balance complexity of combinational logic between registers.

Pipelined signals have been suggested by Zhao and Gajski [ZG05]. It is interesting to note that they specifically state that while they see the need for such a language feature, the could not locate a suitable HDL to embed it in.

FPGAs typically provide two types of *storage elements*: distributed single-bit memories

and addressable memory blocks (of varying sizes). The former is ideally suited to implement fine-grained distributed architectures, but is very inefficient from a logic-resource point of view. The latter uses available chip area very efficiently, but requires sequential read and write operations (in the case of dual-port RAM, they can be parallelized, but multiple read/write cycles have to be performed sequentially). Optimization of memory resource usage is a non-trivial task. It is also an optimization which highly depends on parameters varying with modifications to an algorithm like bit width, number of samples processed, number of parallel channels, interleaving, ...

Current HDLs usually do not support automatic usage of memory blocks (also called memory inference - automatic extraction of memory blocks from functional HDL code). To date, memory blocks need to be instantiated by vendor-provided libraries. By instantiation, full control over suitable selection of size, bitwidth and access pattern is off-loaded to the designer. New versions of the IEEE VHDL RTL Synthesis standard [IEE04] improve upon this situation by defining coding templates for RAM inference. This however is just portable RAM instantiation, omitting the vendor specific libraries.

While there are some attempts to improve memory inference from HDLs, their success is necessarily limited because current HDLs make the respective code analysis extremely difficult.

Especially designs which do not need to access all local data simultaneously (complex state machines, pipelined designs) and designs featuring identical memory access patterns on distributed data sets (multi channel designs) can greatly benefit from a working memory optimization and inference scheme.

Weinhardt and Luk have shown rules for memory inference from software loops [WL99]. These rules apply - to some extent - also to generic HDL designs. A more complete overview on memory optimization techniques for embedded systems was published by Panda et al. [Pan01]. There is however no discussion how such optimizations could be supported by suitable language features. So called *single-assignment* languages have shown that analysis of memory usage can be greatly simplified.

Failsafe circuits are of paramount importance to radiation-exposed digital designs in HEP because digital designs suffer from single-event upsets (SEUs) caused by ionizing particles when traversing the integrated circuit. For state machines, one possible concept is redundancy combined with voting. For data integrity, error-checking codes are often employed. These measures are in principal decoupled from a circuit's primary functionality. Hardware-centric modelling features require however mixture of the two, which results in complex and difficult to change models.

A language feature defining blocks as redundant and failure-aware allows automatic generation of multiple instances. It would allow automatic integration of failsafe-rules like minimum block distance on the die, separate clocks, guard rings or frequent checks by external watchdog circuits[2].

Usually HDLs assume use of binary representation for natural numbers and use of two's complement representation for integers. A more abstract range-based arithmetic data type could allow selection of the underlying representation, including error-checking codes. Checking for data integrity could then be added as separate operation or as automatic

[2]So called *radiation-tolerant* FPGAs feature circuits continuously comparing the FPGAs configuration with a Flash-based reference.

feature.

Such failsafe language features would allow evaluation of design variations trading design size and clock frequency against robustness in the case of SEUs.

The suggested language features can certainly be supplemented by many more features useful to HEP (and also to other domains). Suitable extensions need however be identified in mutual discussion between hardware engineers and physicist.

4.3 On the Importance of Open-Source HDL Implementations

Open-source is often used synonymously to express free access to a project's source code (in contrast to closed-source) and project's granting users the right to use and distribute modified versions of the original project (in contrast to property rights remaining with the original authors). Open source for EDA has been discussed recently by Sud et. al. [SC05] and Carballo [Car05] with special emphasis of education and commercial intellectual property respectively. We believe that fostering both aspects can help improving some tasks in design of digital systems for HEP. In the following, we will motivate this believe and identify the relevant stages and required tools in the design flow.

A solid framework for development of DSPing designs in HEP is not yet available and some special requirements suggest, that even future commercial tools might not match specific HEP needs. It appears vital in such a phase of evolving design methodologies to facilitate further research. This requires researchers to have access to source-code of relevant tools. It seems therefore vital to the HEP community to support an open-source approach in hardware design where needs can not be expected to be satisfied by future standard tools.

VHDL and Verilog HDLs and accompanying silicon compilers (synthesizers) have reached a level of standardization and quality of results which qualifies them as best solution for the last step in the design process (synthesizing hardware from a low-level RTL description).

SystemC has emerged as a viable solution for system simulation which designers in HEP should take advantage of wherever possible.

Design abstraction and generation of custom models is however a weak point of current tools. This is where HEP should look into open-source solutions and try to further understanding of HEP-specific needs and possibilities to satisfy them.

It seems therefore important for future designs in HEP to support an open-source approach in modelling languages which combine concise problem description and mapping into digital hardware as well as into different domain-specific simulation models.

Outlook

Models for functional verification and processing of large data sets do generally not require cycle accuracy. Simulation speed could therefore be increased tremendously by providing untimed models from circuit descriptions. Usually untimed models are used for evaluation and are later refined to actual circuit descriptions. When using untimed models for

functional verification of circuit descriptions, however, it is vital, that the untimed models are bit-accurate with respect to the hardware implementation. This is most important for processing of large data sets, where results will be misleading if arithmetic functionality is not identical. Therefore untimed functional models for verification should be derived automatically from circuit descriptions.

Interface design to allow seamless integration of models within their target environment is a major task. Typically the interface is specified in RTL and recoded every time the interface is used. The lack of of a formal specification renders proofing compliance with a given set of interfaces often impossible.

A more general approach than RTL to provide generic interface descriptions could ease integration and minimize errors. Transaction-level modelling might prove a very efficient vehicle for portable interface descriptions. Klingauf and Kunzel have shown how TLM-based interface descriptions can efficiently be mapped into hardware [KG05].

For *register-based control interfaces*, a straightforward approach to increase documentation quality would be enabled by adoption of generic description schemes like the Register Description Language [Goe06b]. Specification of a set of registers in RDL (or similar languages) would allow automatic extraction of documentation, test protocols and circuit description from one common model.

Chapter 5

Case Study: Language Enhancements to the HDCaml HDL

The scientific contribution of this work lies within the more versatile modelling features and custom simulation models fulfilling specific requirements of High-Energy Physics (HEP). But above all it is the relative ease with which such features can be added to the base implementation if the underlying domain-specific Hardware Description Language (HDL) has a structure suitable for extension.

In chapter 2 HDLs and details on constructing DSLs were presented and discussed. Chapter 3 related general requirements of hardware design with specific challenges in HEP. Chapter 4 suggested an improved design process and required tools for HEP. This chapter will demonstrate extensions to the HDCaml HDL, important to HEP. The existing literature will be presented and it will be shown that the implemented features are not available in similar design environments today. Sample implementations will be presented and data will be given showing the complexity of the respective extensions.

5.1 Selection of HDCaml as Host-language

It is our aim to suggest feasible extensions to HDLs (i.e. whose implementation requires acceptable effort), implementations and evaluation thereof. When selecting a host-language we therefore put emphasis on suitable access to the language/compiler implementation itself, enabling or facilitating implementation of planned extensions. This rules out any closed-source HDL implementation.

To reduce both the initial learning curve and the effort required for implementation of specific language extension, a well modularized base implementation of low-complexity is required. Functional languages have proved to be very efficient for implementing compiler applications [MHS05]. Especially recursive data structures and efficient tree traversal functions reduce the implementation complexity of such applications. We therefore focused on functional implementations of HDLs very early on. While the language used for implementation and the language to be designed are in principle unrelated, it is often the case that the language used for implementation also affects features in the language to be designed. This is also true in the case of HDCaml, which inherits many powerful features from its functional host language OCaml. Although functional languages are considered

rather exotic tools in the hardware design domain, they provide very succinct formulation of many hardware-related modelling issues.

Today, the dominant design strategy in high-performance hardware design is modelling of functionality at RT level, allowing the designer tight control over resulting timing. While higher levels of hardware abstraction potentially provide a better design-space exploration and therefore better results in less time, the related tool effort is important. While there exist tools for specific application domains allowing design entry at higher levels of hardware abstraction, they usually do not cover all needs of a design (as is the case at RTL). Consequently they fragment even further the design process.

I have shown that HEP is an application domain featuring requirements partially out of focus of the mainstream hardware design tools. Therefore I was looking for tools allowing exploration of potential HDL extensions starting from the currently used level of hardware abstraction (namely RTL), yet the potential to improve upon them without a major language redesign effort.

In short, what was required was an open-source HDL implementation, possibly implemented in a functional language and providing hardware abstraction at RTL.

This search coincided with the reimplementation of the experimental open-source Confluence HDL. The Confluence compiler was originally implemented as a stand-alone application written in Objective Caml (OCaml, a pragmatic functional programming language [CM98]). While the compiler approach allowed much freedom in design of language features and notation, it limited its extendability due to the monolithic approach. When this became evident (and a limiting factor to further improvements), the original inventor (Tom Hawkins) decided for a reimplementation by *embedding* the language into OCaml. While the syntax had to be adapted to comply with OCaml syntax, the implementation could take advantage of existing OCaml language features, requiring only implementation of HDL-specific language features. This new language was named HDCaml (Hardware Description in OCaml). My prior experience with Confluence and the matching of the requirements listed above by the new implementation led to adoption of HDCaml as the host language for our planned experimental language extensions.

While HDCaml has proven to be a very versatile base for experimental HDL extensions, we also encountered many shortcomings. Karl Flicker dedicated much work to improving HDCaml's robustness and usability [Fli07]. Daniel Sánchez Parcerisa extended the HDCaml Wiki pages by an comprehensive tutorial [Par06], which triggered many discussions and serves well as a practical reference in daily work.

5.2 VETRA-compliant C++ Simulation Models

The LHC beauty (LHCb) collaboration has developed an extensive collection of software to satisfy the experiment's specific computing needs [LHC06]. VETRA[Szu06] is the data analysis suite developed by the LHCb VErtex LOcator (VELO) collaboration. It is written in C++ and based upon the Gaudi event data processing framework [Bar00] maintained by CERN. VETRA can be fed with both simulated and real data. It allows physics performance characterization of the LHCb VELO sensor and subsequent signal processing. Parts of the signal processing relevant to the VELO's overall performance is implemented on FPGAs situated on the TELL1 Data Acquisition System (DAQ) front-end board [Hae06].

Coarse-grained, manual C++ algorithm implementations integrated in VETRA provide a general impression of an algorithm's physics performance. Once algorithms are to be transferred into digital logic, however, a trade-off between accuracy, resource usage and clock frequency is usually required. Getting a good estimate on the impact of design variations on the overall physics performance is crucial in order to achieve an optimal design. Changes to an implementation can however affect overall physics performance in a way, not necessarily obvious to the designer. The opposite is true, too. Transformations to an algorithm, not affecting the physics performance, can ease or complicate the hardware implementation considerably.

The usual approach is to recode the bit-accurate VHDL functionality in C++ and to perform evaluation using the models in conjunction with VETRA. Keeping two such designs synchronized at bit-level is a very laborious and error-prone task.

Objective

We suggest generating both VHDL- and bit-accurate C++-models from one common source code. This allows for much faster design iteration. Assuming

- availability of tools for mapping model descriptions into a synthesizeable HDLs (specifically VHDL),
- availability of tools for mapping model descriptions into code for simulation, complying to the VETRA coding conventions and
- a representative data sample

a cost function comprising both hardware design parameters and physics performance can be evaluated. This is an enabling factor for true design-space exploration.

VETRA is written in C++ and enforces specific coding guidelines [Cal01]. HDCaml in its standard implementation generates plain C for which wrappers for C++ and SystemC are provided. Neither of these output formats conforms however to VETRA coding guidelines.

In the following, we will give an overview on related approaches. We will contrast original HDCaml-generated C-code with native C++ and VETRA-specific coding requirements. We will show how the HDCaml C-code generator can be modified to comply with VETRA requirements. We will discuss complexity of the required changes and of the resulting code. (We will give an outlook on the range of modifications possible and on the existing limitations.)

Related Approaches

Code generators are usually studied within the framework of compiler construction. When considering the problem in the context of circuit design, special emphasis is on the mandatory support of multiple targets (for implementation and simulation) and on bit- and cycle-accuracy between the generated codes to guarantee modelling coherence. The complexity of a code generator depends heavily on the fitness of the intermediate representation to be transformed into the respective target language. The closer the representation to the semantics of the target, the simpler usually the code generator.

We will review work on generating bit-accurate circuit models in C/C++ with special emphasis of the direct access available to adapt the code generator.

As VHDL and Verilog can be considered the standard input languages for digital hardware synthesis today, we will equate the term *synthesizeable circuit model* with a textual description in either VHDL or Verilog.

As we look for circuit models which can be integrated into existing simulation environments, we need them to be self-executing, i.e. not requiring an external runtime environment. Therefore we use the term *executable circuit model* to describe any self-executing code providing bit-accurate functionality as specified by the respective circuit model. We do not require cycle-accuracy.

With the aforementioned definitions in mind, there are three possible constellations:

1. The executable circuit model is generated from a given synthesizeable circuit model.
2. The synthesizeable circuit model is generated from a given executable circuit model.
3. Both the synthesizeable circuit model and the executable circuit model are generated from a given model.

We will review existing solutions for these three options.

Deriving executable circuit models

The first approach equals to matching the functionality of a given VHDL/Verilog model with some generated C-code.

V2C (VHDL to C translator) is a tool written in 1995 to generate executable C-models from a given VHDL model [Ghe]. It only accepts a subset of VHDL, one limitation being the restriction to data type `bit` and `bit_vector`. Lex and Yacc are used for parsing of the VHDL-model. The sources for V2C are freely available.

VHDL2C by Laurent Prud'hon is a tool to generate C from VHDL[Pru00]. It is written in Java and uses JavaCC (an open source parser generator) to parse a VHDL-subset (using a VHDL-93 grammar provided by Christoph Grimm, University of Frankfurt) and to generate C-code identical in functionality. It was written in 2000 and aims partly at improving upon the limitations of V2C. It supports most of the synthesizeable subset of VHDL including the data types `bit_vector` and `std_logic_vector`.

Both V2C and VHDL2C perform a language conversion, replacing VHDL constructs with corresponding C constructs. While this enables generation of executable C-models, it inhibits usage of more advanced target language features, not mapping directly into basic VHDL constructs (including many code optimization techniques). To enable target code optimizations, it is necessary to provide an intermediate representation of the VHDL source code, which reflects its functionality instead of the VHDL code structure.

The first such tool extracting a control and dataflow graph (CDFG) from behavioural VHDL is called **CHESS** and has been presented by Namballa et. al. in 2004 [Nam03; NRE04]. Their original motivation was extraction of CDFGs from VHDL for usage in high-level synthesis, where CDFGs are the model of choice for combining and evaluating models from different sources. Lex and YACC are used for lexical and syntactic analysis of the VHDL model. Further functionality implemented in C++ transforms the parse tree

obtained from YACC into the final CDFG. The source code of CHESS is not available to the public.

Deriving synthesizeable circuit models

The second approach equals to translating a given C-model into VHDL/Verilog. C inherently lacks the capability to express parallelism, therefore one of the main tasks of any such tool is to extract sequences of commands which can be executed in parallel. C also lacks the notion of arbitrary-length bit vectors and therefore syntesizeable circuit descriptions extracted from C have either to adopt bit widths of native C-types (basically 8,16,32) or require in-depth inspection of code to deduce more optimal bit widths from the given C-code. (See also the article by Edwards [Edw06] for a thorough discussion on fundamental issues in C-based synthesis.)

Therefore a one-to-one mapping between C-code and VHDL/Verilog is unfeasible. One either has to extend the C-language with suitable data types and expressions of parallel execution or one has to give some freedom to the tool to approximate the given code with a generated synthesizeable circuit model.

Language extensions to the C/C++ programming language allowing a bit-accurate extraction of a synthesizeable circuit model from at least a subset of the language are for example Handel-C [Cel04a] or SystemC [IEE06].

Approximation of required bit width from an algorithmic description is the domain of *floating-point to fixed-point tools*. While there are many tools generating optimized code for existing DSP architectures (implying constant operand bitwidth), there are only few considering each operand's bitwidth separately, as is suitable for digital systems design. One example of such a tool is the FRIDGE framework [Ked98].

Deriving executable and synthesizeable circuit models

The third approach is more generic as it allows to choose any language appropriate for circuit description. It then derives executable and synthesizeable circuit models choosing the most suitable target language respectively (we require C++ and VHDL). While the former two approaches always use one language for two purposes (VHDL/Verilog for modelling and implementation in the first case; C/C++ for modelling and simulation in the second case), this approach does not artificially limit circuit design flexibility by enforcing modelling in either of the target languages. It allows usage of the best fitting language for each domain.

CebaTech has developed the **C2R Compiler** which uses untimed C for design specification and exploration. It then generates fast cycle-accurate C and synthesizable Verilog RTL. CebaTech C2R Compiler is a commercial product and closed-source. Therefore we have no access to its cycle-accurate C generator.

BlueSpec developed the **BlueSpec Compiler** which uses SystemVerilog, extended with BlueSpec specific design directives for circuit specification. Term Rewriting Systems are used for internal circuit representation. From this common source, fast cycle-accurate C and synthesizable Verilog RTL models are generated. BlueSpec Compiler is a commercial product and closed-source, therefore we have no access to its cycle-accurate C generator.

Occam is a parallel programming language, originally designed as the native language for transputers. Extensive know-how has been collected during the 1990s on compiling Oc-

cam into FPGAs [PL91; PC00a; PC00b, see also chapter 2.4.2]. Roger Peel has presented an advanced Occam-to-FPGA compiler written in Java [PI05]. The parsing is based on SableCC, an object-oriented framework for compiler generation. The output of SableCC is further processed to obtain an optimized Netlist representation (in EDIF format) for direct download into Xilinx FPGAs. The compiler is not publicly available. While the project itself does not provide generation of executable models in C, there are a number of Occam-to-C crosscompilers available (Kent Retargetable occam Compiler - KRoC, The Amsterdam Compiler Kit - ACK). The approach distinguishes itself from the preceding ones by the fact that it uses a parallel programming language as input, whose properties and related transformations have been investigated extensively. It has been observed that many Occam structures map efficiently into hardware.

The **Quiny SystemC Front-End** allows SystemC-to-VHDL code transformation [SN06]. This is achieved by replacing the original SystemC library with the Quiny library. When the model is executed, it outputs pre-defined VHDL-macros for the hierarchical structure (modules, ports, processes), the functionality (process sensitivity lists, accessed ports, statements and expressions) and the used types (ports, signals, variables, ...). The Quiny library is publicly available. Generating VHDL from SystemC models using Quiny is comparable to deriving C-models from HDcaml models. It differs however by the fact that Quiny does not use an intermediate circuit representation, it is rather a macro extension library than a compiler or embedded language.

HDCaml C-model Style Deficiencies

The LHCb C++ coding conventions [Cal01] enforce specific rules to be followed. In the following we will contrast these requirements with code style of standard HDCaml C models as generated by `Systemc.output_model`. Details of the generated HDCaml C-models are given in appendix A.2.

The first and most obvious difference between the LHCb coding conventions and HDCaml generated C-models is that the former requires C++ code to be used, while HDCaml generates models in pure ANSI-C.

The first step therefore requires transforming the used functions and data types into C++ compliant class members and containers. More specific, the following steps have to be performed to reformulate models in an object-oriented manner:

- generating classes,
- converting functions to methods,
- replacing calls to functions by references to methods.

Another issue is the mapping of bit vectors in arrays of type `unsigned long`. While this is not in contradiction to the LHCb coding conventions, it requires many awkward and error-prone workarounds for bit vectors exceeding 32 bits and using two's complement representation. It would be more convenient to use a library providing arbitrary-bitwidth vector types and respective operations.

The C model maps all internal signals into an anonymous array, providing named pointers only for a subset of signals (inputs, outputs, named wires), thereby greatly limiting debugging.

Implementation

The original HDCaml C-model generator provides a data structure enabling setting of input- and reading of output-ports. A function `cycle_simulator` propagated changes on the inputs as corresponding to one clock cycle in a sequential circuit.

The new C++-model generator extends the original functionality by a data structure, providing full access to all signals in the circuit, enabling efficient debugging. This is especially desirable in view of future derived SystemC model generators, allowing to take advantage of SystemC integrated development environments for debugging. All code is ISO C++ compliant and does therefore fulfill the foremost requirement of the LHCb C++ coding conventions.

In the original HDCaml C-model generator, bits and bit vectors were mapped into arrays of `unsigned long`. This required cumbersome and error-prone handling functions to guarantee arbitrary-width bit vectors. It also obscured the final model's code. The C++ model generator takes advantage of the `ac_int` library by MentorGraphics [Tak06], providing data types and respective operators for signed and unsigned bit vectors of arbitrary width. These data types have been designed to outperform and replace SystemC's data types. It can therefore be expected that eventual future SystemC models can use the same or similar data types.

Implementation Complexity

The original HDCaml C-model generator (`systemc.ml`) requires about 700 lines of code. The reimplementation to generate ISO C++ compliant code (`cpp.ml`) requires only 310 lines of code. The reduction is mostly due to the additional used `ac_int` library. The code is easily accessible and any future adaptation to varying C++ coding conventions is expected to require only minor modifications. This includes changes to comply with SystemC coding conventions.

HDCaml Module	Lines of Code
systemc.ml	712
cpp.ml	310

Table 5.1: Lines of code for HDCaml C and C++ model generators

Example

C-models of different algorithms employed by the LHCb VELO subdetector were derived from their Confluence (the predecessor of HDCaml) models. Original data was recorded using prototypes of the VELO detector. To integrate the C models into the existing Vetra simulation framework alterations to the model had to be made by hand to comply to LHCb C++ coding conventions.

While integration of the C models into Vetra could be achieved and correct simulation could be shown [MS06], the manual intervention limited design-cycle time greatly. The required modifications to the output generator such that generated code would have complied to LHCb C++ coding conventions were considered rather complex, due to the internal structure of the Confluence compiler.

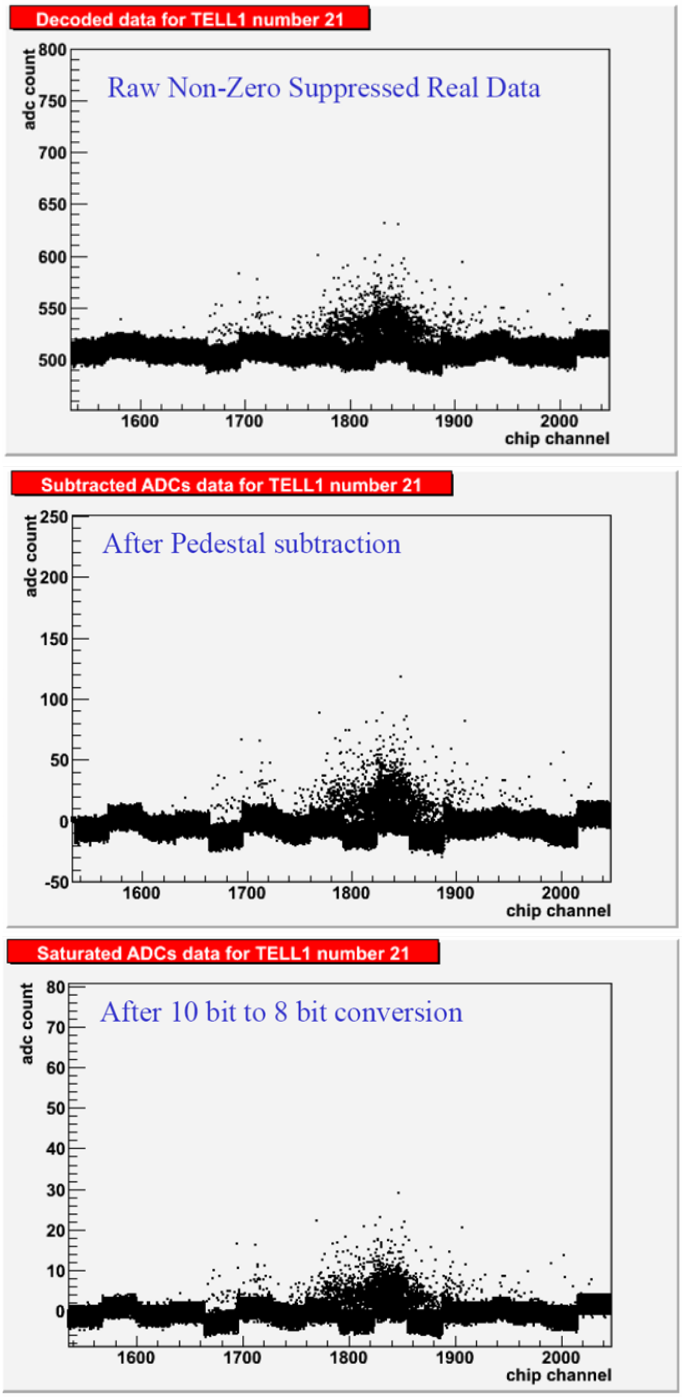


Figure 5.1: LHCb VELO data before and after processing by VETRA simulation framework using automatically generated code blocks.

Figure 5.1 shows Vetra-generated output of raw data before and after processing. The code employed for signal processing was derived from Confluence models and adapted manually to comply to LHCb C++ coding conventions.

Due to the tight schedule of detector commissioning in 2007 prior to startup of the Large Hadron Collider (LHC), the experiments could not be repeated with the respective HDCaml generated code. However, the HDcaml generated C++ model was compared to the Confluence-generated C model and was found to produce identical results.

Evaluation

The HDCaml VETRA-compliant C++ model generator presented provides a solution to generate models from HDCaml models which can be integrated into VETRA without manual intervention to the generated code. It thereby removes the need to maintain manually a VETRA simulation model for evaluation of cost function parameters requiring execution of VETRA. This is a big step toward provision of qualified feedback from hardware design to the designers of the respective HEP experiment. It also adds flexibility to hardware design as experiment designers can give qualified feedback to a number of hardware design variations.

However, the produced models are still cycle-accurate RTL models and interfacing them with the surrounding VETRA data structures remains a tedious (although only singular) task to be accomplished manually.

5.3 Variable-bitwidth Arithmetic

When representing numbers in digital hardware, there are two basic choices:

- fixed-point representation
- floating-point representation

Fixed-point representation provides fixed accuracy over the whole given number range, while floating-point representation allows a much larger number range at the cost of variable accuracy and more complex implementation of arithmetic operations.

When designing digital circuits where available logic resources are limited, fixed-point representation is often chosen for its cheaper implementation. GPPs or DSPs are usually designed around a fixed operand bitwidth. When implementing arithmetic operations in programmable logic, however, it is possible to chose a different bitwidth for every implemented operation, thereby balancing hardware resource usage and quantisation noise.

Bitwidth optimization of arithmetic circuits is a wide field in itself and there exist many different optimization algorithms. It is therefore mandatory for any HDL allowing for bitwidth-abstraction to provide an interface to external tools. An implementation of such an interface in HDCaml, allowing export of dataflow-graphs, will be presented in chapter 5.4.

In the following, we will present related approaches and a bitwidth-aware extension to the HDCaml HDL. We will discuss complexity of the extension and give examples to demonstrate its usage.

Objective

A proper design environment should allow bitwidth-independent modelling followed by a refinement of the bitwidths used where necessary. To implement such a design strategy, a bitwidth-independent fixed-point data type is required. The same data type should accept specification of bit widths, but should not require them. Where no bit width is given, operators should *propagate* the respective information such that range and accuracy are adapted as necessary to accomodate results.

While the importance of bitwidth-optimization is well known, there is currently no HDL supporting this design approach at register-transfer level.

Related Approaches

There are many design environments at different levels of abstraction for helping designers to implement Digital Signal Processing (DSPing) algorithms in hardware. In the following, we will review both the hardware-abstraction and domain-specificity provided by some of these systems.

As pointed out before, we are interested in bit width abstraction at Register-Transfer Level (RTL). There are many tools operating at higher levels of hardware-abstraction. Although we will review such tools and their data-abstraction at the highest level, we will always point out what data-abstraction is still available at register-transfer level. This distinction is useful, as only modelling at RTL enables direct control over a design's critical timing.

One can differentiate between *language-based* and *blockset-based* approaches. The former provides a language for problem description and a set of tools which takes this description as input. The latter provides the user with a set of predesigned blocks, most often allowing modification of some generic parameter within a certain range. These blocks can be interconnected to create new functionality.

While block-based approaches usually allow very efficient coverage of functionality in a well defined domain, they lack the flexibility necessary for succinct algorithm expression when the original domain is left or the optimization function differs from the one originally foreseen. Where a wide or undefined application domain is targeted, language-based approaches are therefore clearly to be preferred.

VHDL

VHDL in its current version [IEE02] does not support any fixed-point data types. VHDL does however provide the `numeric_std` standard-logic-vector numeric operations package, which defines the datatypes `unsigned` and `signed` whose bitwidth can be chosen by the user at declaration.

The main limitation in designing generic, bitwidth-independent functionality in VHDL is the requirement of range bounds to be *locally-static*. *Locally-static* expressions are expressions whose value can be evaluated at analysis of the respective design unit [IEE02, p.113]. This is in contrast to *globally static* expression whose value can only be evaluated after elaboration of the respective design unit.

Due to this requirement, functions can not calculate any of their parameter's bitwidths depending on other parameter's data (like e.g. a unary operator its output bitwidth as

a function of the input bitwidth). Parameter bitwidths need to be computed outside the function implementing the intended functionality. This requires repetition of functionality as the determination of the best fitting bitwidth is usually tightly coupled with the performed operation.

Doulus has made available a VHDL package allowing description of fixed-point data types in 2003 [Bro03]. This package adds information of the binary-point position to existing VHDL data types.

There exists a suggestion for introducing a fixed-point type with the next update of the VHDL standard. An implementation for demonstration has been made available as demonstration package `fixed_pkg` [Bis06]. It uses negative index values to denote fractional positions (i.e. `sfixed (2 downto -2)` stands for a 5bit-wide signal with three integer (index 2..0) and two fractional (index -1..-2) bits). Listing 5.1 demonstrates basic usage of the suggested data types. Observe that the declaration of signal `atimesb` at line 4 explicitly needs to state the bitwidth which has to be computed using the functions `sfixed_high` and `sfixed_low`.

For both packages applies that, while they ease expression of algorithms using fixed-point number representation, they can not revoke VHDL's *locally static* requirement for range bounds. Therefore the same splitting of functionality as described above is still required to provide bitwidth-independent functionality (using a little bit more compact notation, though), practically rendering bitwidth independent model construction infeasible.

```
1 [...]
2 Signal a      : sfixed (5 downto -3); -- 6 integer, 3 fractional bits
3 Signal b      : sfixed (7 downto -9); -- 8 integer, 9 fractional bits
4
5 Signal atimesb : sfixed (sfixed_high (a,'*',b) downto
6                          sfixed_low  (a,'*',b));
7 Begin
8   atimesb <= a * b; -- signed fixed point multiplication
9 [...]
```

Listing 5.1: Suggested fixed-point data type in VHDL-200X

SystemC

SystemC features several fixed-point data types [IEE06, p.276] (`sc_fixed`, `sc_ufixed`, `sc_fxval`). Therefore fixed-point data types can be used in SystemC models.

`sc_fixed` and `sc_ufixed` are fixed-precision types, requiring specification of their bitwidth at declaration (implying the same restriction as the requirement of locally static range bounds in VHDL). While this type enables bit-accurate modelling, it inhibits bitwidth-independent algorithm specification.

The type `sc_fxval` is a variable-precision type, meaning that a variable of this type can hold values of different (arbitrary) precision over its lifetime.

As there exists no definition of a synthesizeable SystemC subset, support of data types for synthesis differs between tools. Current situation seems to be that only tools specialized on DSP designs aim at supporting fixed-point data types. While there is an effort by the Open SystemC Initiative (OSCI) to agree upon a generally accepted synthesizeable SystemC subset (SSC), the relevant document [OSC04] has made little progress since its

appearance in 2004. In this draft, the support for synthesis of fixed-point data types is defined as mandatory, to comply with the future SSC. The variable-precision type `sc_fxval` is explicitely excluded from the SSC.

Therefore abstraction of models with respect to fixed-point data types hardly varies between VHDL when using the packages discussed above and SystemC.

Handel-C

Handel-C is a small subset of C, extended with some constructs to provide guidance to hardware generation and to express parallelism [Cel04a]. Handel-C is the input language to the DK tool-suite by Celoxica. DK aims at extracting both hardware and software from a given specification in Handel-C. DK is a HW/SW-codesign tool.

Handel-C itself does not provide support for fixed-point types. Celoxica ships however a fixed-point library with its DK tool. The library provides a data structure (`FIXED_SIGNED (intWidth, fracWidth)`), interpreted as fixed-point type and a number of operations acting on the provided fixed-point structure. While integer- and fractional bitwidth can be specified, they need to be constant at compile-time [Cel04b, p.7]. Due to Handel-C's restriction to C, the library can not implement operator overloading (which is only available in C++) but has to use normal functions even for basic arithmetic operations (FixedAdd(), FixedSub(), ..).

```
1  FIXED_SIGNED(intWidth, fracWidth) Fixed;
2  // This declaration defines the following structure:
3    //struct {
4    //signed intWidth FixedIntBits;
5    //    Width of integer part of the fixed-point structure.
6    //    Must be positive and a compile time constant.
7    //signed fracWidth FixedFracBits;
8    //    Width of fraction part of the fixed-point structure.
9    //    Must be positive and a compile time constant.
10   //};
```

Listing 5.2: Definition of fixed-point data type in Handel-C

```
1  #include <fixed.hch>
2  set clock = external "P1";
3  typedef FIXED_UNSIGNED(4, 8) MyFixed;
4  void main(void)
5  {
6  MyFixed fixed1, fixed2, fixed3;
7  // Give the fixed-point number value 3.25
8  fixed1 = FixedLiteralFromInts(FIXED_ISUNSIGNED, 4, 8, 3, 64);
9  // Give the fixed-point number value 4.75
10 fixed2 = FixedLiteralFromInts(FIXED_ISUNSIGNED, 4, 8, 4, 192);
11 // Add the numbers together
12 fixed3 = FixedAdd(fixed1, fixed2);
13 }
```

Listing 5.3: Usage of fixed-point data type in Handel-C

Impulse C

Impulse C [Imp06] aims at moving algorithms expressed in ANSI-C to an FPGA for algorithm acceleration. It provides data types to represent signed and unsigned fixed-point numbers of 8, 16 or 32bit [Bod06]. Fractional bitwidth can be chosen by the designer. Macros are provided for basic arithmetic operations (+, *, /) and for conversion to integer and floating-point types. Macros only accept operands having matching integer and fractional bitwidths. The designer is responsible for correct scaling of the operands and has to specify explicitly the fractional bitwidth when calling the macro. When targeting hardware, the CoDeveloper tool can generate RTL-models in either VHDL or Verilog.

Impulse C and its tool chain does not support any arbitrary-bitwidth fixed-point data type at any level of hardware abstraction.

Catapult Synthesis

Catapult Synthesis [Men07] is a high-level synthesis tool by Mentor Graphics Inc. It accepts ANSI C++ input from which optimized RTL designs are generated. Design exploration is guided through constraints specified by the User. SystemC models are generated for verification and VHDL or Verilog models for synthesis. Catapult seems to work best on data flow-centric applications [Dee06].

While Catapult allows designers to describe designs in C++, which does not provide any fixed-point data type, Catapult does use fixed-point data types to model different design variations. These fixed-point data types have been made available as a separate C++ library free of charge [Tak06]. The library is said to be designed with special emphasis on high execution speed (a common problem with SystemC fixed-point data types). Using these libraries allows bit-accurate modelling, it does however not provide the means for bitwidth-independent modelling.

Internal working details of Catapult are not available and prices starting at $140,000 for a one-year license make evaluation infeasible.

Forte Cynthesizer

Cynthesizer from Forte Design Systems is very similar to Catapult, but focuses on design specifications in SystemC rather than C++. The tool generates RTL-models using a SystemC transaction-level-model (TLM) design description as input.

Cynthesizer generates RTL models from SystemC models containing `sc_fixed` or `sc_ufixed` types. As it does not provide synthesis support for `sc_fxval`, the principal modelling limitations mentioned earlier for SystemC still apply. It is not clear to what extent Cynthesizer can extract fixed-point models from algorithms expressed in pure C, using float or double types.

Synplify DSP

Synplify DSP software [Syn07] is a high-level DSP synthesis tool accepting specifications in form of Simulink models. It performs architectural optimizations and produces synthesizeable RTL models in either VHDL or Verilog. The ModelSim blockset allows automatic

propagation of fixed-point variable's parameters (bitwidth, binary point position). Therefore Synplify DSP enables bit width independent modelling of blocksets.

AccelDSP

AccelDSP [Xil07] is a high-level MATLAB-language based tool by Xilinx for designing DSP blocks for Xilinx FPGAs. The tool automates floating- to fixed-point conversion, generates synthesizeable VHDL or Verilog, creates a testbench for verification and a fixed-point C++ model for accelerated simulation. AccelDSP does not provide any means to model functionality at RTL.

HDFS

HDFS [Ray06] (Hardware design in F#, pronounced "F sharp") is a reimplementation of HDCaml in F# [Mic06]. It inherits all of HDcaml's features, but simplifies compiler implementation by exploiting advanced features of F#. There is a fixed-point library for HDFS which does however not allow positioning of the binary point outside the signal's bit range, thereby limiting efficient representation of scaled value ranges. However, HDFS provides a type system comparable to HDCaml's and thereby enables similar features.

Summary

Support for synthesis of fixed-point data types is currently improving by the month. Reviewing the current state of support for fixed-point data types can therefore only be considered a snapshot.

One can differentiate two different approaches:

1. *Hardware-centric* solutions provide fixed-point notation, but basically retain a problem description in hardware-relevant terms. All reviewed solutions require the user to specify low-level details like bitwidth and binary-point position for each variable.

2. *Algorithmic-centric* solutions provide means to describe algorithms in an environment typically used for algorithm development (C, Matlab) using data types not being concerned with operand bit width. These solutions typically generate optimized (according to some cost function) fixed-point designs from a given floating-point specification.

While the former approach allows more direct control of the generated hardware, the inflexible type systems currently inhibit bitwidth-independent problem description. The latter approach allows bitwidth-independent problem description but does not provide means for direct control of generated hardware. An exception is AccelDSP, which allows automatic propagation of fixed-point variables' properties, but only in the framework of a blockset-based design environment.

It seems that our approach provides for the first time support for synthesizeable arbitrary-width fixed-point data types in a functional HDL. The library available for HDFS is very similar, but only appeared after the library presented in this work.

Implementation

The HDCaml module `fixp.ml` provides a fixed-point data type and respective operators and handling functions. As with all HDCaml signals, the bitwidth needs only to be specified for inputs or constants, all further bitwidth information is automatically propagated through operators and functions applied to the signals to the output ports. The fixed-point data type `fixp_signal` is obtained by supplementing HDCamls basic `signal` data type by a binary point position and a sign information. The binary point position is given relative to the right of the LSB, moving towards the MSB with increasing values (i.e. a binary point position of 0 denotes an integer, one of 2 denotes two fractional bits). The binary point can also be positioned outside the bit range of the signal, enabling efficient modelling of scaled value ranges. Conversion functions are provided between conventional HDCaml signals and fixp signals. One can instantiate native fixp inputs and outputs as well as constants (approximated from a given value of type `float`).

Replacement operators are provided for all of HDCaml's arithmetic operators (`+`, `-`, `*`, `==`, `/==`, `<`, `<=`, `>`, `>=`). All operations retain full accuracy, i.e. the resulting signal's bitwidth is extended to accommodate additional bits.

The module in itself is rather simple. Its power comes from combining HDCaml's principle of signal type propagation with a syntax suitable for expressing fixed-point algorithms. This enables succinct formulation of flexible models. Because information like binary-point position is only given once and propagated automatically subsequently, models are kept smaller and can adapt if this information is changed (in contrast to models in conventional HDLs, where redundant information needs to be given at many points, specifically in and out ports of functions).

A second advantage, building upon these capabilities is the possibility to build sophisticated functions, encapsulating generic knowledge on optimization of fixed-point circuits. See the smart divider presented below for an example.

Implementation Complexity

The `fixp.ml` module is a classical *data type extension* with accompanying operators and handling functions. It hides complexity from the user by providing a suitable notation for fixed-point variables and respective operations. All functionality is directly mapped into basic HDCaml signals, operators and functions. Figure 5.2 gives the lines of code of `fixp.ml`.

HDCaml Module	Lines of Code
fixp.ml	310

Table 5.2: Lines of code for the HDCaml fixp.ml module

The major drawback of the current implementation of `fixp.ml` is that the fixed-point information is lost in the internal HDCaml circuit representation. The consequence is that no output generator or optimization applied to the intermediate representation can take advantage of this information. The current implementation allows however to distribute `fixp.ml` as a stand-alone module, without the need of a tight integration into the HDCaml base implementation.

Example

Smart Constant Divider

This is a small example, showing the advantage of combining fixed-point notation with a type system allowing type propagation through functions.

Division is a costly operation in hardware. Many algorithms exist for deriving efficient implementations of this operation depending on the available hardware resources.

If the divisor is known in advance (constant), the respective inverse can be precomputed. Often the precision of the inverse is limited to reduce required hardware resources. This is identical to converting the inverse to a fixed-point representation and varying the used number of fractional bits. The number of fractional bits used determines the accuracy of the result.

A fractional bit n_i positions to the right of the binary point has a weight of 2^{-n_i}. Therefore truncation of the signal at this position causes a maximum error of $e < 2^{-n_i+1}$ if the bit (and all bits of lower significance) was set. The required fractional bitwidth n_f to obtain an error smaller than e can be computed by $n_f > ld(e)$. The total maximum error of the division is then given by $e_{div} < 2^{1+n_f+\lceil ld(a_{max})\rceil}$. Therefore to obtain a total error smaller than a given value e_1, the required bitwidth of the inverse is $n_f = ld(e_1) - (1 + \lceil ld(a_{max})\rceil)$. This only applies if all trailing bits are set. If this is not the case, additional unset fractional bits can be truncated without reducing the accuracy.

Table 5.3: Impact of fractional bitwidth on fixed-point representation accuracy

Bits used	Binary representation	Decimal representation	Error
$\rightarrow \infty$	.0000 0101 0001...	$\rightarrow$0.02	$\rightarrow$0
12	.0000 0101 0001	0.019775390625	0.0000415
8	.0000 0101	0.01953125	0.0004688
7	.0000 010	0.015625	0.004375
6	.0000 01	0.015625	0.004375

If the language used for modelling does not provide notation for fixed-point representation, multiplication by the inverse maps into an integer multiplication followed by a shift operation. No information on accuracy and binary point position can be extracted from the model. The designer is responsible for calculating the correct values and keeping track of correct implementation.

Reusable models however should emphasize error rather than bitwidth and consequently a versatile division function should accept a divisor and acceptable error as input, computing the required bitwidth (of both the inverse and the output) by itself. HDCaml together with `fix.ml` enables encapsulating of algorithms for bitwidth estimation and provides binary-point information to connecting functional blocks. It can also be enhanced to give additional information to the user guiding him in his choice of acceptable error.

The HDCaml function `div_const` takes the divisor and a maximum acceptable division error as input and computes the best matching fixed-point representation of the inverse. It also outputs information on the next lower and higher precision available and respective error figures.

```
1 > ocaml hdcaml.cma fixp.cma
2          Objective Caml version 3.08.1
3
4 # open Hdcaml;;
5 # open Design;;
6 # open Fixp;;
7
8 # start_circuit "Test";;
9 # let a = fixp_input ~signed:false ~bp_pos:10 ~width:20 ~name:"a";;
10 # let a_scaled=mul_const a 0.02 0.01;;
11
12  (mul_const): signal unsigned, bw 20, frac 10
13  (mul_const): factor = 0.020000 (=0b.00000101000111101011...)
14  (mul_const): requested max. error = 0.010000
15  (mul_const): minimum fractional bitwidth required:      17 (Error = 0.003438)
16  (mul_const): next higher-accuracy fractional bitwidth: 19 (Error = 0.001484)
17  (mul_const): next lower-accuracy fractional bitwidth:  15 (Error = 0.011250)
```

Listing 5.4: Demonstration of smart constant divider

Listing 5.4 shows usage of `mul_const` with the OCaml top-level system. Lines 1-8 set up HDCaml, in line 9, a fixed-point input is declared using 10 integer and 10 fractional bits. The resulting signal is multiplied by 0.02 ($=\frac{1}{50}$) and a total maximum error of 0.01 is specified at line 10. The function computes the correct bitwidth (17, see line 15), declares the respective constant fixed-point representation of the factor and provides a signal, holding the result of the multiplication. In line 16 and 17 additional information is provided on further choices in case lower or higher errors would be specified respectively.

LHCb VELO LCMS

The LHCb VELO sub detector [LHC01], delivers data in sets of 32 values. These sets are expected to contain a linear common mode, changing with every set. Therefore an algorithm has been conceived to be implemented in the TELL1 DAQ front-end board [Hae06] to calculate the mean and slope value for each set and to correct for the linear common mode. This signal preprocessing stage is called Linear Common Mode Suppression (LCMS). Guido Haefeli et al. have reviewed different algorithms and their respective impact on FPGA resource usage [BHK01].

The current implementation of the LCMS algorithm is based on a VHDL model and an accompanying C simulation model. In VHDL, it is especially difficult to retain generic modelling while optimizing the implementation for minimal resource usage and high throughput. We have therefore looked into possibilities to model LCMS in a generic way, yet allowing for low-level modifications necessary to guarantee optimum solutions. While LCMS accepts and outputs integer values, the internals of the algorithm are tightly coupled to fixed-point variables of varying accuracy and range. The `fixp.ml` allows succinct formulation of the respective operations and avoids obscuring the code by fixed-point related optimizations. It does however still enable direct access to hardware implementation and respective bitwidth of signals where necessary. With such a model, exploration of the design-space becomes much easier than with the respective VHDL model.

Figure 5.2 shows the basic steps of the LCMS algorithm.

There are two major areas affected by changes to the LCMS algorithm: FPGA resource usage and physics performance. Reducing internal accuracy can save hardware resources,

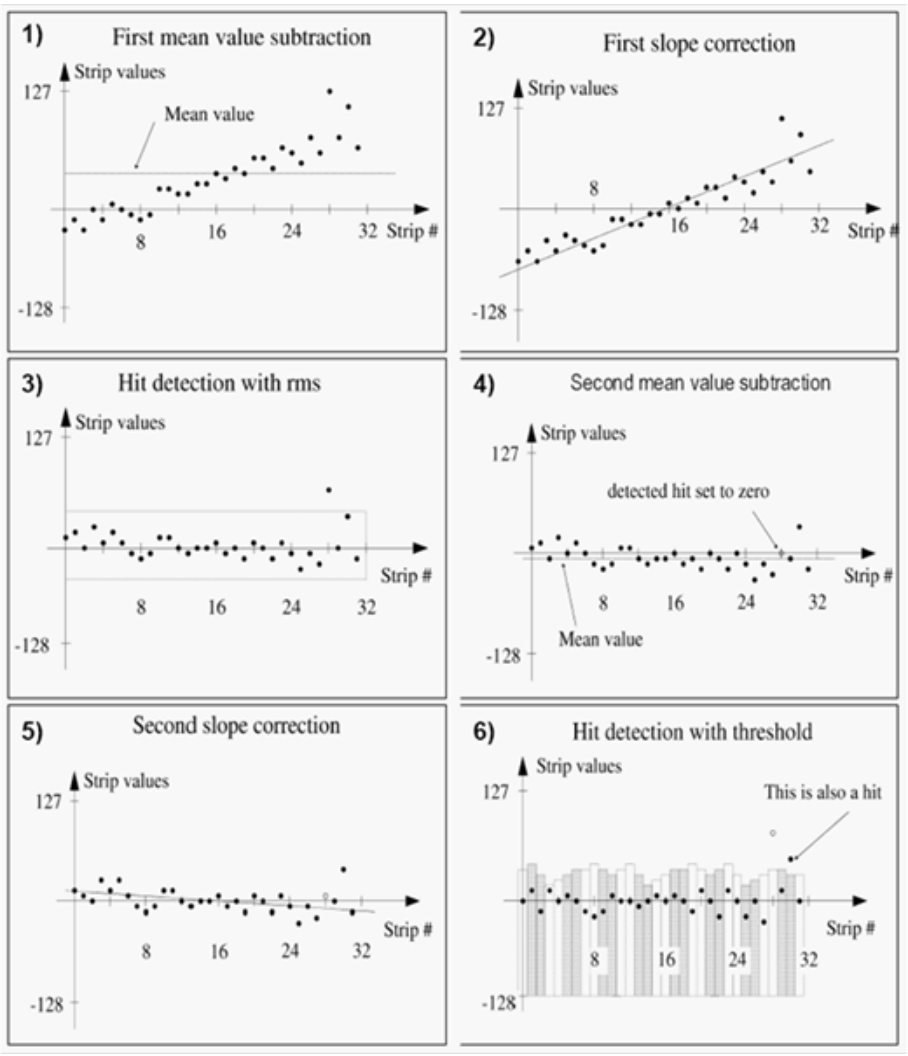


Figure 5.2: Example of the LCMS algorithm being applied to a set of 32 values.

Design	f_{max}	LUTs			9bit DSP	Mem Bits
	[MHz]		Reg only	Logic only		
Integer precision (14.0)	67	1830	1392	220	8	0
Medium precision (14.5)	66	1886	1414	236	20	0
Full precision (14.10)	65	1898	1426	228	20	0

Table 5.4: FPGA resource usage of LCMS design variations

but might affect physics performance. Efficient modelling using the fixed-point data type as described before enables generation of a set of design variations in short time to be evaluated subsequently. Automatically generated VETRA-compliant C++ models as described in section 5.2 facilitate evaluation of the latter. The VHDL model can be used to obtain matching digital logic resource usage. For this example we will only evaluate digital logic resource usage.

The two big blocks contributing to the overall resource usage of the LCMS algorithm are the delay buffers (FIFOs) and the arithmetic units computing mean and slope. The current HDCaml implementation does not support efficient memory modelling for VHDL (i.e. current VHDL synthesizers can not infer memories from the generated VHDL models). We therefore focus on mean and slope calculation only, which are also the most suitable blocks for demonstration of fixed-point modelling.

The required accuracy of the slope highly depends on the actual common mode to be found in the aquired data. If the common mode is highly linear, a high precision might give improved results. If however a linear slope turns out to be only a coarse approximation of the common-mode, it might be worth trading slope accuracy for additional hardware resources for further correction algorithms. As there are two stages of mean and slope calculation in LCMS, it might also be worth exploring the advantage of different slope accuracies in the two stages.

Table 5.3 shows the results of design variations altering the precision of the slope signal in both stages. Resource usage was obtained by synthesizing the HDCaml-generated VHDL model with Altera Quartus 6.1 targeting a Stratix S30 device. While the increase in used LUTs and the decrease of the maximum design frequency shows a continouos relation with the bitwidth (caused by the additionally required combinational logic), the used DSP blocks show a sudden increase. The latter is caused by operations exceeding the 9- or 18-bit boundaries. It becomes clear from the resource usage data that the full precision design comes at little cost when compared to the medium precision design. It is however very expensive in hardware terms to move from the integer precision design to the medium or full precision design.

Evaluation

The major advantage in using fixed-point signals for modelling of the LCMS algorithm lies in an improved separation of algorithm and hardware implementation (signal bitwidth). The design variations listed above have been obtained by adding a single command, limiting the amount of fractional bits used for representation of the slope. All further related changes are performed automatically by type propagation.

5.4 Graph Description Language (GDL) Models

Control and Data Flow Graphs (CDFGs) are an important abstract representation of hardware designs for visualization, optimization, transformation and design-space exploration. There exist efficient tools for optimization of different design aspects and graph visualization. While construction of a CDFG is a natural (usual manual) step in design refinement, extraction of CDFGs from existing formal HDL models is less common. The former is however mandatory if design iterations, based on already existing HDL models want to use existing high-level tools. A CDFG can also serve as an alternative description of an existing design, easing visibility of certain features otherwise obscured in a textual description.

As we have identified fast design iterations as an important need in the HEP hardware design process, we are most interested in providing a suitable path for CDFG extraction from existing HDCaml models.

We will review related approaches, demonstrate CDFG extraction from HDCaml models suitable for display using the aiSee tool and will contrast the respective module's implementation complexity with other solutions.

Related Approaches

While there are tools abundant accepting some kind of textual graph description as input, there exist only few tools able to extract graph descriptions from formal models. Vallerio et al. [VJ03] present a tool to extract task graphs from embedded system descriptions in C. Ramballa et al. [NRE04] build upon this work, but address designs at a lower level of abstraction, by extracting CDFGs from VHDL models. The Seoul National University's design automation laboratory provides a tool for extracting CDFGs from VHDL [CJA05]. Optimizations are subsequently applied to the CDFGs and results can be made available as VHDL or C models. All projects include a component to parse the original model (C or VHDL) to build an intermediate representation.

There exists no unique definition of CDFGs (see Orailoglu and Gajski [OG86] for an early discussion on CDFGs; Wu et al. [Wu02] a CDFG description suitable for Hardware/-Software Codesign (allowing modelling of resolved signals)). A recent suggestion extending on CDFG are behavioural network graphs (BNG) [Ber02] enabling both high-level transformation and efficient resource usage estimation.

There is no generally accepted textual format for CDFGs. For graphical representation of graphs, the VCG tool [San96] (and its successor aiSee) is frequently used. aiSee by AbsInt GesmbH is provided free of charge for academic research and accepts input descriptions in the *Graph Description Language* (GDL). GDL represents a graph by two distinct sets, one describing all nodes and one describing directed edges between given nodes.

Implementation

The new HDCaml module `gdl.ml` allows output of graph descriptions in GDL from any HDCaml circuit description. `gdl.output_cdfg` maps the circuit's elements and wires into CDFG nodes and edges. The resulting files are ready for display using the aiSee tool. Some extra information is added to format the graphs such that they resemble typical schematic RTL circuit descriptions (using GDL formatting features).

Specifically

- The information flows from top to bottom,
- Edges contain only vertical and horizontal sections and have a tree-like layout,
- The respective bit width is displayed for each multi-wire edge,

HDCaml's basic functional building blocks map directly into most CDFG's elements. Table 5.5 defines the used node types as generated by the HDCaml module `gdl.ml` and their graphical representation as visualised by aiSee. The complete set of HDCaml building blocks is listed in the third column, giving a direct relation between each building block and the respective node type.

Edges represent directed flow of information without delay. Each edge is annotated with its bitwidth (unnamed edges) or signal name and bitwidth (named edges).

Implementation Complexity

All related approaches presented in section 5.4 require processing of textual input (VHDL or C). This includes complex compiler-like parsing, analysis and transformations. Our approach relies on the fact that execution of any HDCaml program generates an internal representation of the described circuit, available to all other HDCaml modules (including the GDL output generator). The extra effort required to extract a CDFG is therefore reduced considerably by accessing directly the intermediate representation.

Functional languages are especially suited for implementation of compiler-like tasks like tree construction and traversal due to their frequent support of cyclic recursive structures and powerful means to traverse them. Therefore the use of a functional language (OCaml in our case) facilitates further the implementation.

The internal HDCaml representation maps naturally into basic elements of a CDFG, as shown in table 5.5. Therefore only minimal transformations are required.

Table 5.6 gives the lines of code of the module, as determined by ocamlwc [Fil01]. The low linecount reflects both the reduced complexity of mapping the intermediate data structure into CDFG representation and of the efficient coding accomplished using OCaml. Obviously modifications to such a small module are easy to perform.

Some limitations on the graph layout are caused by the lack of a suitable GDL feature to define the position of an edge head relative to its target node (i.e. one can not enforce an edge to end to the left of the second line of its target node; This is however possible for origins of edges). For some cases, a workaround is used, reversing the edge's direction and drawing the arrowhead towards the source. Future versions of GDL/aiSee might improve upon this situation.

Example

A simple counter circuit serves as basic example to demonstrate visualization of circuits. The counter has two inputs: `en` (enable/disable counter) and `up` (count up/down), a register holding the current value, two arithmetic units performing addition and subtraction and one multiplexer selecting either of the result for input to the register. Figure 5.3 shows the result when using the generated CDFG in GDL format as input to the aiSee 3.0 Software.

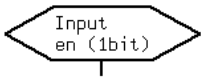
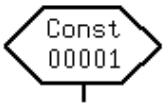
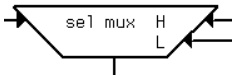
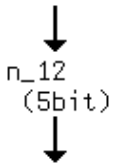
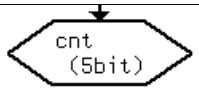
Node type	Graphical representation	HDCaml basic elements
Input		`Signal_input`
Constant		`Signal_const`
binary Operation		`Signal_and`, `Signal_or`, `Signal_xor`, `Signal_eq`, `Signal_lt`, `Signal_add`, `Signal_sub`, `Signal_mul`, `Signal_mul_u`, `Signal_concat`
unary Operation		`Signal_not`, `Signal_select`
Choose Value		`Signal_mux`
Register		`Signal_reg`
unnamed edge		all unnamed dependencies
named edge		`Signal_signal`
Output		`Sink_output`

Table 5.5: Control and data flow graph elements as generated by the HDCaml module gdl.ml and as visualised by aiSee.

HDCaml Module	Lines of Code
gdl.ml	145

Table 5.6: Lines of code for the HDCaml gdl.ml module as given by ocamlwc

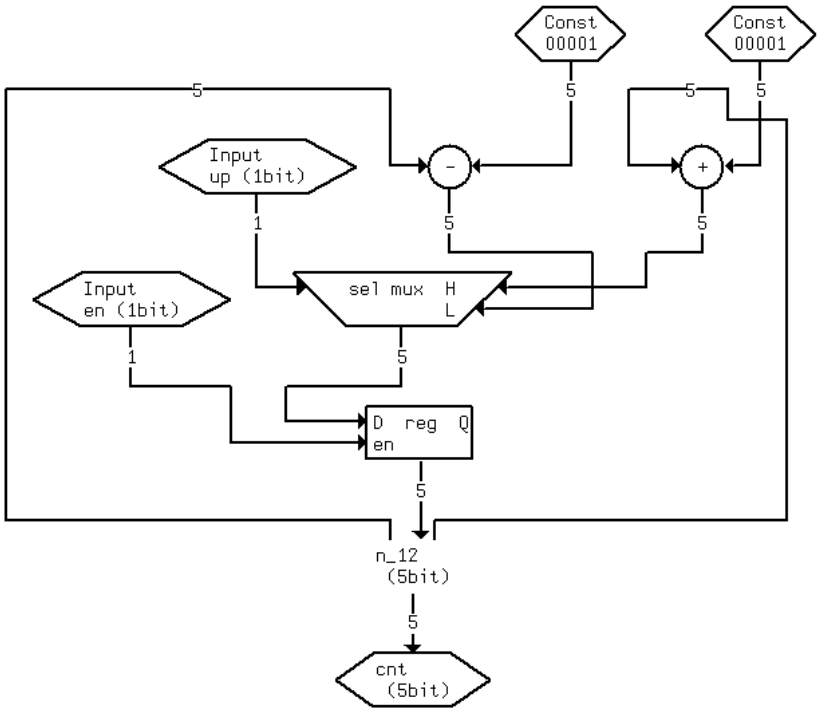


Figure 5.3: Visual representation of a CDFG derived from a HDCaml model using gdl.ml and displayed using aiSee 3.0.

Evaluation

It is straightforward to adapt the module to other output formats, as long as it uses a set representation (nodes, edges) of the graph. Due to the limited coding required, even a full rewriting of a similar module for different output formats is feasible within short time. As the CDFG is directly derived from the intermediate representation, the module is not affected by changes to the grammar or the syntax of HDCaml, as long as the internal representation is not affected.

5.5 Evaluation

Chapters 5.2, 5.3 and 5.4 have presented extensions to the HDCaml HDL, implementing generation of VETRA-compliant C++ models, bitwidth independent fixed-point data types and generation of GDL models respectively. The provided functionality is generally not available in contemporary EDA tools. In cases where similar functionality is available, the respective tool requires considerable implementation effort (typically including parsing and lexical analysis).

The presented HDCaml extensions, in contrast, all show very low implementation complexity (as expressed by required lines of code). This is due to the fact that they can take advantage of the existing HDCaml language infrastructure including free access to all intermediate stages of model construction.

The low complexity of the two custom model generators were enabled by the direct access to HDCaml's intermediate model representation. The intermediate circuit representation itself is a recursive data structure for which OCaml provides very efficient notation and handling functions (both in terms of coding and execution speed). The low implementation complexity therefore is fundamentally owed to the open-source approach of HDCaml (direct access to compiler internals) and its implementation using a functional language (recursive data structures and handling functions).

The bitwidth independent fixed-point data type derives its simple implementation from HDCaml's signal type propagation mechanism, facilitated by OCaml's type inference mechanism and the fact that the fixed-point type maps naturally into original HDCaml basic types.

For all three examples it is the availability of functional features derived from OCaml as a host language and the accessibility of the HDCaml implementation which enables low complexity of respective implementations.

The central point when evaluating HDCamls implementation, which sets it apart from current HDL implementations, is its intermediate model representation and the ease with which it can be accessed. This architecture allows separation of functionality (model entry, model-to-model transformation, model-to-target transformation).

The fundamental limitation of HDCaml is the inability to extend its internal representation. Therefore extension to the language can be implemented with ease, as long as its functionality is compatible with the existing intermediate representation. If this is not the case, implementation effort becomes important. (One - currently not possible - example being the conservation of fixed-point types from model-entry through intermediate representation to custom model output.)

Chapter 6

Conclusion and Outlook

This work has discussed the history of Hardware Description Languages (HDLs) and their importance to the hardware design process. Expanding the scope to language construction in general, Domain-Specific Language (DSL) design and implementation issues were reviewed. Subsequently lessons learned from DSL design and implementation were related to HDLs design and implementation.

The design of digital systems for High-Energy Physics (HEP) experiments was analyzed and special needs were identified. It was found that there exist HEP-specific needs (highly domain specific models, automated generation of custom simulation models) not covered by existing tools and methodologies. A sample of recent HEP designs showed that the issue is very real and affecting large-scale designs.

I have suggested to adopt an open-source approach to tools potentially requiring addition of HEP-specific features. This includes HDLs used in HEP experiments. I have argued that functional languages are superior in terms of implementation efficiency and have supported this argument by providing a case study of several extensions to the HDCaml HDL, using a functional language as its host language. Complexity of related existing implementations and of HDcaml extensions were discussed. HDCaml extensions show a considerably lower implementation complexity while integrating HEP-specific needs into the language implementation (in contrast to add-on tools).

HDCaml together with its HEP-specific extensions provides a framework for integration of tasks in HEP digital systems design executed separately before. It provides unprecedented design-space exploration capabilities by enabling evaluation of multi-domain cost functions. This is achieved by providing multiple custom simulation models and the *necessary access to tune them for future needs.*

The HDCaml HDL can act as a tool to evaluate new HDL language features without the need to redesign a new HDL from scratch. For many extensions, it would however be favorable to have a more versatile intermediate circuit representation. Behavioural network graphs [Ber02] might prove a sensible choice. More advanced tranformation rules could then be explored for design-space exploration and circuit optimization. Efficient memory allocation could be such a domain open to many further improvements, once represented in a suitable format.

Using a real-world example for evaluation of Confluence and HDCaml (the LHCb VELO LCMS algorithm) has proven extremely helpful in that real needs of users were identi-

fied very early on. It has however shown as well the many implications of a high-speed algorithm-centric signal processing design, seldomly considered when concentrating on a small part of a HDLs. Many foreseen experiments could not be conducted because of the tight timing and limited resources.

Outlook

With the availability of platforms to integrate new language features, identification of domain-specific abstractions should potentially gain traction. Such abstraction should subsequently allow formulation of more succinct and readable models, narrowing the communication gap between the worlds of application domain experts and hardware designers. They could also lead to improved verification at lower levels of abstraction (like VHDL or even at chip-level) as model knowledge could be propagated using e.g. PSL.

Recent publications like [ZG05] show that very basic features (seen from a signal processing perspective) are only now being suggested as HDL language features. Zhao and Gajski [ZG05] even state explicitly that no suitable platform for integration of the suggested features could be found. It seems therefore likely that further additional features to HDLs will be suggested and discussed in the near future as HDLs are used for modelling of increasingly complex algorithm implementations. This process will be important to HEP.

Design of intellectual property (IP) blocks is an area of application for domain-specific HDLs, not discussed in this work. IP blocks are usually configured by the user but subsequently fail to deliver an optimized overall system due to the black-box approach they take, limiting optimization to the scope of each block. Design of such blocks usually means creation of a (VHDL) code generator which is a tedious task and could be improved by use of a more suitable language. Therefore there is a considerable need for HDLs enabling more generic models in IP design. If the language used provided the necessary problem-abstraction to perform optimizations on a level above IP-block level, the result could be superior to current solutions.

Appendix A

HDCaml Implementation Details

This appendix will give detailed information on selected parts of the HDCaml 0.2.10 implementation. Language extension as described in the main text are built upon these base implementation.

An in-depth discussion of more HDCaml internals and improvements (especially on debugging) implemented in release 0.2.10 can be found in [Fli07].

To read the following subsections, a certain knowledge on OCaml notation might, while not essential, certainly be helpful.

A.1 Internal Circuit Representation

The circuits described using HDCaml functions are represented internally in a recursive data structure of type:

```
type circuit =
 | Circuit of id * string * circuit list * signal list * sink list.
```

Each circuit consists of a list of signal inputs and transformations (`type signal`) and a list of signal sinks (`type sink`). The list of signal inputs and transformations is a linked list, representing the circuit by means of HDCaml basic building blocks and connectivity between them. Following any signal path through the structural description will finally lead to a node of type `sink`. All functions attempting to extract some output description from the internal circuit description just need to traverse backwards the list of signals, taking the elements of the list of sinks as starting points.

Only information contained in the type `signal` is available to any output generator function. The full definition of the type `signal` is given in listing A.2.

```
1  open Hdcaml;;
2  open Design;;
3
4  start_circuit "MyCircuit";
5    (* some HDCaml circuit description functions*)
6
7  let myCircuit = get_circuit() in
8    Systemc.output_model myCircuit;
9    Verilog.output_netlist myCircuit;
10   Vhdl.output_netlist myCircuit
```

```
11 ;;
```

Listing A.1: HDCaml circuit representation

```
1  type signal = | Signal_input of id * string * width
2  |             Signal_signal of id * string * width * signal
3  |             Signal_const  of id * string
4  |             Signal_empty  of id
5  |             Signal_select of id * int * int * signal
6  |             Signal_concat of id * signal * signal
7  |             Signal_not    of id * signal
8  |             Signal_and    of id * signal * signal
9  |             Signal_xor    of id * signal * signal
10 |             Signal_or     of id * signal * signal
11 |             Signal_eq     of id * signal * signal
12 |             Signal_lt     of id * signal * signal
13 |             Signal_add    of id * signal * signal
14 |             Signal_sub    of id * signal * signal
15 |             Signal_mul_u  of id * signal * signal
16 |             Signal_mul_s  of id * signal * signal
17 |             Signal_mux    of id * signal * signal * signal
18 |             Signal_reg    of id * signal * signal
```

Listing A.2: Definition of type signal

A.2 HDCaml C-code Generator

HDCaml provides a function `Systemc.output_model` to generate a bit- and cycle-accurate C-model of the described circuit. The generated files provide a stand-alone simulator with direct access to all inputs, outputs and intermediate signals. This simulator can be used for fast processing of large data sets or for assessing the accuracy of a specific hardware implementation within a software simulation framework.

All inputs, outputs and internal signals are represented as arrays of `unsigned long`. The least significant bit (LSB) of the signal equals the LSB of the lowest element of the array.

If the signal has more bits than a single element can hold[1], another element is added to the array[2].

A sequential logic circuit consists of two basic functional building blocks: storage elements and combinational logic. The HDCaml C-code generator maps all storage elements present in a circuit into an array `memory` of type `unsigned long`. Combinational logic is implemented by reading values from `memory`, applying the respective function and writing the results back into `memory`.

The generated data structure `simulator_s` combines the array `memory` with a structure of pointers for all signals, pointing at the respective position in `memory` each.

The generated function `init_simulator` resets all elements in `memory` to their initial values and sets all pointers, such that they point to the correct position in `memory`.

[1] On most computers, the unsigned long data type is 32-bit wide

[2] For signals exceeding 32 bits, the first element would therefore hold bits 0-31, the second element bits 32-63 and so on

The generated function `cycle_simulator` implements the combinational logic for the whole circuit. Therefore a basic simulation loop looks like:

```
write circuit inputs
call cycle_simulator
read circuit outputs
```

The function `Systemc.output_model` writes a set of three files:

- `<Design Name>.c` (the code to simulate the design)
- `<Design Name>.h` (the corresponding header file)
- `<Design Name>_sc.h` (a wrapper for SystemC)

The generated code consists of three parts:

- struct simulator_s is a data structure providing access to the circuit's inputs, outputs and all (named and unnamed) internal signals.
- a list of public functions to set up the simulation (`new_simulator`, `delete_simulator`, `init_simulator`)), a convenience function to search for ports `find_simulator_port` and the function to advance the simulation by a single clock cycle `cycle_simulator`.
- a set of private functions used by `cycle_simulator` to implement the actual functionality of the design (`hdcaml_mask`, `hdcaml_not`, `hdcaml_and`, `hdcaml_xor`, `hdcaml_or`, `hdcaml_concat`, `hdcaml_concat_simple`, `hdcaml_select`, `hdcaml_select0`, `hdcaml_select_inword`, `hdcaml_eq`, `hdcaml_lt`, `hdcaml_add`, `hdcaml_sub`, `hdcaml_mul`, `hdcaml_mul_u`, `hdcaml_sign_extend`, `hdcaml_mul_s`, `hdcaml_mux`, `hdcaml_reg`, `hdcaml_reg_update`, `hdcaml_assert`).

`get_circuit()` maps all basic HDCaml operators into basic nodes of the internal data structure of type `circuit`, which has subtypes corresponding to each basic HDCaml operator. When `Systemc.output_model` analyzes the internal data structure, it can generally map every single basic node directly into one basic C-function (or a set of a few).

```
1  #ifdef __cplusplus
2  extern "C" {
3  #endif
4
5  // Simulator Data Type
6  struct simulator_s {
7    struct {
8      struct {
9        // inputs:
10       unsigned long * en ; // input en : 1 bits , 1 words
11       // outputs:
12       unsigned long * cnt ; // output cnt : 5 bits , 1 words
13       // wires:
14       unsigned long * n_11 ; // wire n_11 : 5 bits , 1 words
15       unsigned long * low ; // wire low : 1 bits , 1 words
16       unsigned long * high ; // wire high : 1 bits , 1 words
17       unsigned long * gnd ; // wire gnd : 1 bits , 1 words
18       unsigned long * vdd ; // wire vdd : 1 bits , 1 words
19       // wires in subcircuits:
```

```
20      } Counter;
21    } signals;
22    unsigned long memory[10];
23 };
24
25 typedef struct simulator_s *simulator_t;
26
27 // Simulator Constructor
28 simulator_t new_simulator();
29
30 // Simulator Destructor
31 void delete_simulator(simulator_t);
32
33 // Simulator Initialization
34 void init_simulator(simulator_t);
35
36 // Find signal in simulator structure
37 typedef struct _signal_t { unsigned long *signal; unsigned long width; } signal_t;
38 signal_t find_simulator_port(simulator_t sim, char *name);
39
40 // Simulator Cycle Calculation
41 void cycle_simulator(simulator_t);
42
43 #ifdef __cplusplus
44 }
45 #endif
```

Listing A.3: counter.h - HDCaml-generated C header file

List of References

[Ang06] Angoletta, Maria E. "Digital Low Level RF". In: *European Particle Accelerator Conference EPAC'06*. 2006. URL: http://cdsweb.cern.ch/record/971791.

[Ang03] Angoletta, Maria E. "Digital Signal Processing in Beam Instrumentation: Latest Trends and Typical Applications". In: *6th European Workshop on Beam Diagnostics and Instrumentation for Particle Accelerators DIPAC 2003*. 2003. URL: http://cdsweb.cern.ch/record/624433.

[Ang05a] Angoletta, Maria E., et al. "Beam Tests of a New Digital Beam Control System for the CERN LEIR Accelerator". In: *Proceedings of the Particle Accelerator Conference PAC 2005*. 2005. URL: http://cdsweb.cern.ch/record/844032.

[Ang05b] Angoletta, Maria E., et al. PS Booster Beam Tests of the new Digital Beam Control System for LEIR. AB-Note-2005-017. CERN-AB-Note-2005-017. Tech. rep. Geneva: CERN, 2005.

[Bai99] Baines, J. T. M., et al. Pattern Recognition in the TRT for the ATLAS B-Physics Trigger. ATL-DAQ-99-007. Tech. rep. Geneva: CERN, 1999. URL: http://cdsweb.cern.ch/record/683897.

[Bar06] Barney, D., et al. Implementation of On-Line Data Reduction Algorithms in the CMS Endcap Preshower Data Concentrator Card. CMS Note. Tech. rep. CERN, 2006. URL: http://cdsweb.cern.ch/record/1000411.

[Bar00] Barrand, G., et al. "GAUDI: The software architecture and framework for building LHCb data processing applications". In: *International Conference on Computing in High Energy and Nuclear Physics, CHEP 2000*. 2000. 92–95. URL: http://cdsweb.cern.ch/record/467678.

[BD93] Barton, D. L., and D. D. Dunlop. "An introduction to MHDL". In: *Microwave Symposium Digest, 1993., IEEE MTT-S International*. Vol. 3. 1993. 1487–1490. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=276859.

[Bar95] Barton, D. "Advanced modeling features of MHDL". In: *Proceedings of the International Conference on Electronic Hardware Description Languages*. 1995. URL: http://citeseer.ist.psu.edu/barton95advanced.html.

[BHK01] Bay, A., G. Haefeli, and P. Koppenburg. LHCb VELO off detector electronics preprocessor and interface to the level 1 trigger. LHCb-2001-043. Tech. rep. Geneva: CERN, 2001. URL: http://cdsweb.cern.ch/record/691717.

[BH98] Bellows, P., and B. Hutchings. "JHDL-an HDL for reconfigurable systems". In: *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*. 1998. 175–184. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=707895.

[Ber02] Bergamaschi, R. A. "Bridging the domains of high-level and logic synthesis". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 21.5 (2002). 582–596. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=998629.

[Bis06] Bishop, David. *Fixed point package user's guide*. 2006. URL: http://www.vhdl.org/vhdl-200x/vhdl-200x-ft/packages/Fixed_ug.pdf.

[Bje98] Bjesse, Per, et al. "Lava: hardware design in Haskell". In: *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM Press, 1998. ISBN 1581130244. DOI: 10.1145/289423.289440. 174–184. URL: http://portal.acm.org/citation.cfm?id=289423.289440.

[Bod06] Bodenner, Ralph. *Fixed-Point Arithmetic in Impulse C*. 2006. URL: http://www.impulsec.com/IATAPP106_FIXEDPT.pdf.

[Bro03] Bromley, Jonathan. *Synthesisable fixed-point package fix_std*. 2003. URL: http://www.doulos.com/knowhow/vhdl_designers_guide/models/fp_arith/fix_std_0.2.zip.

[Cal01] Callot, Olivier. Revised C++ coding conventions. LHCb-2001-054. Tech. rep. Geneva: CERN, 2001. URL: http://cdsweb.cern.ch/record/684691.

[Car05] Carballo, J. A. "Open HW, open design SW, and the VC ecosystem dilemma". In: *System-on-Chip for Real-Time Applications, 2005. Proceedings. Fifth International Workshop on*. 2005. 3–6. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1530905.

[CS78] Case, G. R., and J. D. Stauffer. "SALOGS-IV: A Program to Perform Logic Simulation and Fault Diagnosis". In: *Design Automation, 1978. 15th Conference on*. 1978. 392–397. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1585203.

[Cel04a] Celoxica. *Handel-C Language Reference Manual*. RM-1003-4.2. 2004. URL: http://www.celoxica.com/techlib/files/CEL-W0410251JJ4-60.pdf.

[Cel04b] Celoxica. *Platform Developer's Kit - Fixed-point Library Manual*. RM-1021-1.0. 2004.

[Cha76] Chappell, S. G., et al. "Functional simulation in the LAMP system". In: *DAC '76: Proceedings of the 13th conference on Design automation*. New York, NY, USA: ACM Press, 1976. DOI: 10.1145/800146.804793. 42–47.

[CJA05] Choi, Kiyoung, Jinhwan Jeon, and Yong-Jin Ahn. *Control-Data Flow Graph Toolset Homepage*. 2005. URL: http://poppy.snu.ac.kr/CDFG/cdfg.html.

[Chu65] Chu, Yaohan. "An ALGOL-like computer design language". In: *Communications of the ACM* 8.10 (Oct. 1965). 607–615. ISSN 0001-0782. DOI: 10.1145/365628.365650.

[CM98] Cousineau, Guy, and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998. ISBN 0521576814. URL: http://dx.doi.org/10.2277/0521576814.

[Dec07] Decaluwe, Jan. *MyHDL Home Page*. 2007. URL: http://myhdl.jandecaluwe.com.

[Dee06] DeepChip. *Catapult C Synthesis Feedback*. 2006. URL: http://www.deepchip.com/items/else06-07.html.

[Deh06] Dehning, Bernd, et al. "The LHC Beam Loss Monitoring System's Surface Building Installation". In: *Proceedings of the 12th Workshop on Electronics For LHC and Future Experiments* (Sept. 2006). URL: http://cdsweb.cern.ch/record/1020105.

[Dep82] Department of the US Air Force. *Draft Request for Proposal F33615-83-R-1003, VHSIC Hardware Description Language (VHDL)*. Department of the US Air Force, 1982.

[DD68] Duley, J. R., and D. L. Dietmeyer. "A Digital System Design Language (DDL)". In: *Computers, IEEE Transactions on* C-17.9 (1968). 850–861. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1687472.

[Edw06] Edwards, S. A. "The Challenges of Synthesizing Hardware from C-Like Languages". In: *IEEE Design & Test of Computers* 23.5 (2006). 375–386. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1704728.

[Eis99] Eisenbiegler, Dirk. "Ein Kalkül für die Formale Schaltungssynthese". PhD thesis. University of Karlsruhe, 1999. URL: http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=/1999/informatik/1\&search=/1999/informatik/1.

[EB99] Eisenbiegler, Dirk, and Christian Blumenröhr. "Gropius - Advanced Reuse Concepts in a New Hardware Description Language". In: *Proceedings of REUSE'99*. 1999. URL: http://citeseer.ist.psu.edu/632023.html.

[Fil01] Filliâtre, Jean-Christophe. *ocamlwc - A program to count the lines of code and documentation in OCaml sources*. 2001. URL: http://caml.inria.fr/cgi-bin/hump.en.cgi?contrib=409.

[Fli07] Flicker, Karl. "Analyse und Erweiterung der Hardwarebeschreibungssprache HDCaml". MA thesis. Technische Universität Graz, 2007.

[FMW05] Foster, Harry, Erich Marschner, and Yaron Wolfsthal. "IEEE P1850 PSL: The Next Generation". In: *Proceedings of the Design & Verification Conference, 2005 (DVCon 2005)*. 2005. URL: http://www.pslsugar.org/papers/ieee1850psl-the_next_generation.pdf.

[GK83] Gajski, D. D., and R. H. Kuhn. "New VLSI Tools". In: *Computer* 16.12 (1983). 11–14. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1654264.

[Gaj96] Gajski, Daniel D. *Principles of digital design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN 0133011445. URL: http://portal.acm.org/citation.cfm?id=248673.

[Ghe] Ghezzi, Cristian. *V2C: VHDL to C translator*. URL: http://web.tiscali.it/sito01/pro/v2c/main.htm.

[Gho99] Ghosh, Sumit K. *Hardware Description Languages: Concepts and Principles*. Wiley-IEEE Press, 1999. ISBN 0780347447. URL: http://portal.acm.org/citation.cfm?id=519673.

[Goe06a] Goering, Richard. "MyHDL - Scripting language takes silicon turn". In: *EE Times* (Jan. 2006). URL: http://eetimes.com/news/design/showArticle.jhtml?articleID=177101584.

[Goe06b] Goering, Richard. "Register language available through open source". In: *EE Times* (Mar 2006). URL: http://www.embedded.com/showArticle.jhtml?articleID=181503468.

[Gra03] Grastveit, G., et al. "FPGA Coprocessor for High-Level Trigger Applications". In: *9th Workshop on Electronics for LHC Experiments LECC 2003*. 2003. URL: http://cdsweb.cern.ch/record/722065.

[GDG04] Gupta, Sumit, Nikil D. Dutt, and Rajesh Gupta. *Spark: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer, 2004. ISBN 978-1-4020-7837-8.

[Hae06] Haefeli, G., et al. "The LHCb DAQ interface board TELL1". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 560.2 (May 2006). 494–502. DOI: 10.1016/j.nima.2005.12.212.

[HDB99] Hammes, Jeffrey P., Bruce A. Draper, and Willem A. P. Böhm. "Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems". In: *: Computer Vision Systems: First International Conference, ICVS'99, Las Palmas, Gran Canaria, Spain, January 1999. Proceedings.* 1999. 83+. URL: http://www.springerlink.com/content/1b0c54lrbhe5tcqe.

[Haw05] Hawkins, Tom. *Confluence Home Page.* 2005. URL: http://www.funhdl.org/wiki/doku.php?id=confluence.

[Haw06] Hawkins, Tom. *HDCaml Home Page.* 2006. URL: http://www.funhdl.org/wiki/doku.php/hdcaml.

[Hv79] Hill, D., and W. vanCleemput. "SABLE: A Tool for Generating Structured, Multi-Level Simulations". In: *Design Automation, 1979. 16th Conference on.* 1979. 272–279. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1600118.

[HP87] Hill, Frederick J., and Gerald R. Peterson. *Digital Systems: Hardware Organization and Design.* 3rd ed. Wiley, 1987. ISBN 0471808067.

[Hol05] Holzer, E. B., et al. "Beam loss monitoring system for the LHC". In: *Nuclear Science Symposium Conference Record, 2005 IEEE.* vol. 2. 2005. 1052–1056. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1596433.

[Psl] IEEE Standard for Property Specification Language (PSL). tech. rep. 2005. 1–143. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1524461.

[IEE06] IEEE SystemC Standard working group. *IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual.* 2006. 0–423.

[IEE88] IEEE. *1076-1987 IEEE Standard VHDL Language Reference Manual.* IEEE, 1988. ISBN 0-7381-4324-3. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1003477.

[IEE94] IEEE. *1076-1993 IEEE Standard VHDL Language Reference Manual.* IEEE, 1994. ISBN 0-7381-0986-X. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=3116.

[IEE02] IEEE. *1076-2002 IEEE Standard VHDL Language Reference Manual.* IEEE, 2002. ISBN 0-7381-3247-0. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=7863.

[IEE04] IEEE. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. Tech. rep. 2004. 1–112. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1342563.

[Imp06] Impulse Accelerated Technologies Inc. *Impulse C Homepage.* 2006. URL: http://www.impulsec.com.

[JB99] Jennings, James, and Eric Beuscher. "Verischemelog: Verilog embedded in Scheme". In: *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages.* New York, NY, USA: ACM Press, 1999. 123–134. URL: http://dx.doi.org/10.1145/331960.331978.

[Ked98] Keding, H., et al. "FRIDGE: a fixed-point design and simulation environment". In: *Design, Automation and Test in Europe, 1998., Proceedings.* 1998. 429–435. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=655893.

[Kho06] Khomich, Andrei, et al. "Using FPGA coprocessor for ATLAS level 2 trigger application". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 566.1 (Oct. 2006). 80–84. URL: http://www.sciencedirect.com/science/article/B6TJM-4K2T3YP-9/2/6dcdbdce046714b886157cf5847d9dc6.

[Kd91] Kiczales, Gregor, and Jim des Rivières. *The Art of the Metaobject Protocol*. The MIT Press, 1991. ISBN 0262610744.

[KG05] Klingauf, W., and R. Gunzel. "From TLM to FPGA: Rapid prototyping with SystemC and transaction level modeling". In: 2005. 285–286. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1568563.

[Kum96] Kumar, Ramayya, et al. "Formal Synthesis in Circuit Design - A Classification and Survey". In: *FMCAD '96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*. London, UK: Springer-Verlag, 1996. ISBN 3540619372. 294–309.

[LLC99] Launchbury, John, Jeffrey R. Lewis, and Byron Cook. "On embedding a microarchitectural design language within Haskell". In: *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM Press, 1999. ISBN 1581131119. 60–69. URL: http://dx.doi.org/10.1145/317636.317784.

[LMR94] Levia, Oz, Serge Maginot, and Jacques Rouillard. "Lessons in language design: cost/benefit analysis of VHDL features". In: *DAC '94: Proceedings of the 31st annual conference on Design automation*. New York, NY, USA: ACM Press, 1994. DOI: 10.1145/196244.196464. 447–453.

[LHC01] LHCb Collaboration. *LHCb VELO (VErtex LOcator) Technical Design Report*. Technical Design Report LHCb. Geneva: CERN, 2001.

[LHC06] LHCb. *The LHCb computing home page*. 2006. URL: http://lhcb-comp.web.cern.ch/lhcb-comp/.

[Li95] Li, Yanbing. "HML - An innovative Hardware Description Language and its translation to VHDL". PhD thesis. Cornell University, 1995.

[LL00] Li, Yanbing, and M. Leeser. "HML, a novel hardware description language and its translation to VHDL". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8.1 (2000). 1–8. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=820756.

[MSV06] Manthos, N., G. Sidiropoulos, and P. Vichoudis. "An efficient hardware design for rejecting common mode in a group of adjacent channels of silicon microstrip sensors used in high energy physics experiments". In: *IEEE Transactions on Nuclear Science* 53.3 (2006). 1045–1050. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1644987.

[Max03] Maxfield, Clive. "New language makes waves". In: *EETimes.com* (May 2003). URL: http://www.eetimes.com/news/design/columns/max_bytes/showArticle.jhtml?articleID=17408320.

[McC05] McCloud, Shawn. Mentor Graphics European ESL Survey 2005. Tech. rep. Mentor Graphics Corp., 2005. URL: http://www.mentor.com/products/c-based_design/upload/ESL_Survey_Report_EU.pdf.

[McG03] McGettrick, A., et al. "Computer engineering curriculum in the new millennium". In: *Education, IEEE Transactions on* 46.4 (2003). 456–462. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1245168.

[Men07] Mentor Graphics Corp. *Catapult Synthesis*. 2007. URL: http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm.

[MHS05] Mernik, Marjan, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". In: *ACM Computing Surveys* 37.4 (Dec. 2005). 316–344. ISSN 0360-0300. URL: http://portal.acm.org/citation.cfm?id=1118890.1118892.

[Mic06] Microsoft Research. *F# project home page*. 2006. URL: http://research.microsoft.com/fsharp/fsharp.aspx.

[MA01] Mitchell, John C., and Krzysztof Apt. *Concepts in Programming Languages*. Cambridge University Press, 2001. ISBN 0521780985.

[Mue05] Muecke, Manfred. "C/VHDL codesign for LHCb VELO zero suppression algorithms". In: *14th IEEE-NPSS Real Time Conference*. 2005. DOI: 10.1109/RTC.2005.1547399. 156–157.

[MH06] Muecke, Manfred, and Guido Haefeli. "A bitwidth-aware extension to the HDCaml Hardware Description Language". In: *Proceedings of the Forum on specification & Design Languages 2006, FDL'06*. ECSI, 2006.

[MS06] Muecke, Manfred, and Tomasz Szumlak. "Unified C/VHDL Model Generation of FPGA-based LHCb VELO algorithms". In: *12th Workshop on Electronics for LHC and future Experiments (LECC'06)*. 2006. URL: http://cdsweb.cern.ch/record/1034306.

[NRE04] Namballa, R., N. Ranganathan, and A. Ejnioui. "Control and data flow graph extraction for high-level synthesis". In: *Proceedingsof the IEEE Computer society Annual Symposium on VLSI, 2004*. IEEE Computer Society. 2004. 187–192. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1339528.

[Nam03] Namballa, Ravi K. "CHESS: A tool for CDFG extraction and high-level synthesis of VLSI systems". PhD thesis. University of South Florida, 2003.

[O'D02] O'Donnell, John. "Overview of Hydra: A concurrent language for synchronous digital circuit design". In: *Proceedings of the International Parallel and Distributed Processing Symposium 2002 (IPDPS 2002)*. 2002. 234–242. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1016653.

[OG86] Orailoglu, Alex, and Daniel D. Gajski. "Flow graph representation". In: *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*. Piscataway, NJ, USA: IEEE Press, 1986. ISBN 0818607025. 503–509. URL: http://portal.acm.org/citation.cfm?id=318093.

[OSC04] OSCI Synthesis Working Group. SystemC Synthesizable Subset Draft 1.1.18. Tech. rep. Open SystemC Initiative (OSCI), 2004. URL: https://www.systemc.org/download/5/47/73/128/.

[PL91] Page, Ian, and Wayne Luk. "Compiling Occam into field-programmable gate arrays". In: *Oxford Workshop on Field Programmable Logic and Applications*. Ed. by W. Moore and W. Luk. 15 Harcourt Way, Abingdon OX14 1NV, UK: Abingdon EE&CS Books, 1991. 271–283. URL: http://citeseer.ist.psu.edu/98799.html.

[Pan01] Panda, P. R., et al. "Data and memory optimization techniques for embedded systems". In: *ACM Trans. Des. Autom. Electron. Syst.* 6.2 (Apr. 2001). 149–206. ISSN 1084-4309. DOI: 10.1145/375977.375978.

[Par06] Parcerisa, Daniel Sánchez. *HDCaml Tutorial*. 2006. URL: http://www.funhdl.org/wiki/doku.php/hdcaml:tutorial.

[PC00a] Peel, Roger M. A., and Barry M. Cook. "Occam on Field Programmable Gate Arrays - Fast Prototyping of Parallel Embedded Systems.". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, Nevada, USA*. 2000.

[PC00b] Peel, Roger M. A., and Barry M. Cook. "Occam on Field-Programmable Gate Arrays - Optimising for Performance". In: *Communicating Process Architectures - 2000 (WoTUG 23)*. IOS Press, 2000. 227–238.

[PI05] Peel, Roger, and David Pizarro de la Iglesia. "Automatically Generated CSP Provides Verification for Occam-derived Logic Circuits.". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2005, Las Vegas, Nevada, USA, June 27-30, 2005*. 2005. 180–186.

[Pru00] Prud'hon, Laurent. *VHDL to C translator (in french)*. 2000. URL: `http://people.via.ecp.fr/~laurent/dev/vhdl2c/`.

[Ray06] Ray, Andy. *HDFS - Hardware Design in F# project home page*. 2006. URL: `http://code.google.com/p/hdfs/`.

[San96] Sander, Georg. "Visualisierungstechniken für den Compilerbau". PhD thesis. Universität des Saarlandes, 1996.

[Sch01] Schlosser, Joachim. "Development and Verification of fast C/C++ Models for the Star12 Micro Controller". MA thesis. Fachhochschule Augsburg, 2001. URL: `http://schlosser.info/diplomathesis/thesis.html`.

[SN06] Schubert, Thorsten, and Wolfgang Nebel. "The Quiny SystemC Front End: Self-Synthesising Designs". In: *Forum on Specification & Design Languages 2006*. 2006. 135–142. URL: `http://ehs.informatik.uni-oldenburg.de/publications/index.php?action=detail\&id=001378`.

[SR93] Sharp, Robin, and Ole Rasmussen. "Transformational Rewriting with Ruby". In: *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993. ISBN 0444816410. 243–260. URL: `http://portal.acm.org/citation.cfm?id=752235`.

[She84] Sheeran, Mary. "muFP, a language for VLSI design". In: *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*. New York, NY, USA: ACM Press, 1984. ISBN 0897911423. 104–112. URL: `http://dx.doi.org/10.1145/800055.802026`.

[SS] Singh, S., and M. Sheeran. *Designing FPGA Circuits in Lava*. URL: `http://citeseer.ist.psu.edu/258055.html`.

[SK95] Slonneger, Kenneth, and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison Wesley Longman, 1995. ISBN 0201656973.

[SSDM01] Séméria, Luc, Koichi Sato, and Giovanni De Micheli. "Synthesis of hardware models in C with pointers and complex data structures". In: *IEEE Trans. Very Large Scale Integr. Syst.* 9.6 (Dec. 2001). 743–756. DOI: `10.1109/92.974889`.

[SC05] Sud, R., and M. Chaitanya. "Revolution in electronic EDA education/research: GOSPL". In: *Microelectronic Systems Education, 2005. (MSE '05). Proceedings. 2005 IEEE International Conference on*. 2005. 37–38. URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1509353`.

[Syn07] Synplicity. *Synplify DSP Home Page*. 2007. URL: `http://www.synplicity.com/products/synplifydsp`.

[Szu06] Szumlak, Tomasz. *The VETRA home page*. 2006. URL: `http://ppewww.physics.gla.ac.uk/LHCb/Simulation/Vetra.htm`.

[SL73] Szygenda, S. A., and A. A. Lekkos. "Integrated techniques for functional and gate-level digital logic simulation". In: *DAC '73: Proceedings of the 10th workshop on Design automation*. Piscataway, NJ, USA: IEEE Press, 1973. 159–172. URL: `http://portal.acm.org/citation.cfm?id=804011`.

[Szy94] Szymanski, B. K., et al. "Is there a future for functional languages in parallel programming?". In: *Proceedings of the 1994 International Conference on Computer Languages, 1994*. 1994. 299–304. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=288371.

[Tah99] Taha, Walid M. "Multistage programming: Its theory and applications". PhD thesis. 1999. URL: http://portal.acm.org/citation.cfm?id=931031.

[Tak06] Takach, Andres. *Algorithmic C Bit Accurate Integer and Fixed Point C++ Data Types*. 2006. URL: http://www.mentor.com/products/esl/high_level_synthesis/ac_datatypes.cfm.

[Vah03] Vahid, F. "The softening of hardware". In: *Computer* 36.4 (2003). 27–34. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1193225.

[VJ03] Vallerio, K. S., and N. K. Jha. "Task graph extraction for embedded system synthesis". In: *VLSI Design, 2003. Proceedings. 16th International Conference on*. 2003. 480–486. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1183180.

[van77] vanCleemput, W. M. "An hierarchical language for the structural description of digital systems". In: *DAC '77: Proceedings of the 14th conference on Design automation*. Piscataway, NJ, USA: IEEE Press, 1977. 377–385. URL: http://portal.acm.org/citation.cfm?id=809157.

[VR06] Vichoudis, Paschalis, and Serge Reynaud. "A multi-channel optical plug-in module for gigabit data reception". In: *12th Workshop on Electronics for LHC and future Experiments*. 2006. URL: http://indico.cern.ch/contributionDisplay.py?contribId=9\&sessionId=13\&confId=574.

[Vui94a] Vuillemin, J. E. "On Circuits and Numbers". In: *IEEE Transactions on Computers* 43.8 (Aug. 1994). 868–879. URL: http://dx.doi.org/10.1109/12.295849.

[Vui94b] Vuillemin, Jean E. "Fast linear Hough transform". In: 1994. 1–9. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=331821.

[Vui] Vuillemin, Jean. *How to do circuits with Jazz*. URL: http://www.exalead.com/jazz/.

[Web94] Webster. *Webster's Encyclopedic Unabridged Dictionary of the english language*. Gramercy Books, 1994.

[WL99] Weinhardt, Markus, and Wayne Luk. "Memory Access Optimization and RAM Inference for Pipeline Vectorization". In: 1999. 61–70. URL: http://www.springerlink.com/content/rpq35xykvbak1knl.

[Wex89] Wexler, J. *Concurrent Programming in Occam 2*. Ellis Horwood, 1989. ISBN 978-0131617384.

[Wu02] Wu, Qiang, et al. "A hierarchical CDFG as intermediate representation for hardware/software codesign". In: *IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions*. Vol. 2. 2002. 1429–1432. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1179048.

[Xil07] Xilinx. *AccelDSP Synthesis Tool Home Page*. 2007. URL: http://www.xilinx.com/ise/dsp_design_prod/acceldsp.

[Zam06] Zamantzas, Christos. "The Real-Time Data Analysis and Decision System for Particle Flux Detection in the LHC Accelerator at CERN". PhD thesis. Brunel Uninversity, 2006. URL: http://de.scientificcommons.org/17100050.

[ZG05] Zhao, Shuqing, and Daniel D. Gajski. "Defining an Enhanced RTL Semantics". In: *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005. DOI: 10.1109/DATE.2005.111. 548–553.