DISSERTATION

A Tripartite Approach for Design Space Exploration

Hardware/Software Design Space Exploration of Low Volume Embedded Systems

Submitted at the Faculty of Electrical Engineering and Information Technology, Vienna University of Technology in partial fulfillment of the requirements for the degree of Doktor der technischen Wissenschaften (equals Ph.D.)

under supervision of

Univ.Prof. Dr. habil. Christoph Grimm
Institute number: E384
Institute of Computer Technology

and

Univ.Prof. Dr.rer.nat. Ulrich Schmid
Institute number: E366
Institute of Sensor and Actuator Systems

by

DI(FH) Peter Brunmayr
Matr.Nr. 0527426
Universumstrasse 26/12, 1200 Vienna

Date ______________________________

Abstract

Embedded systems are electronic devices, which are integrated into a larger system and optimized for a specific purpose. Depending on the purpose, different design goals such as the maximum costs, maximum power consumption or minimum performance can be identified. Since these goals may be different for each system, particular design methods, which support the designer to make the best design decisions, are required. A difficult and critical decision is the selection of the system architecture and the mapping of functional components to processing units. To find the best solution to this decision, methodologies for exploring the design space have been developed in recent years. These methodologies enable the evaluation and comparison of different architectures and mappings.

Existing solutions, which support the designer during this step, target especially the design of system on chips. However, for low volume systems the design of a new integrated circuit is not profitable. In this case, the demanded functionality is typically realized with standard components. Additionally, most approaches have modeling restrictions. Examples are complex data structures, which are not supported by many current solutions. This significantly limits the level of abstraction.

In this work a new approach for design space exploration, which separately models computation, communication and data structures, is presented. This separation simplifies the generation of various hardware/software implementations and additionally, enables the support of complex data structures. They are mapped to the respective target architecture by using a library based approach. The presented concept targets especially low volume systems. Methodologies are provided, which support the efficient mapping of abstract system models to common standard components. The approach simplifies the evaluation of different hardware/software realizations. It helps the designer to find the best application-to-architecture mapping. The concept has been evaluated via a case study. Different realizations of a Voice-over-IP engine have been generated and compared. The analysis of the modeling effort has shown a significant reduction compared to a traditional design approach.

Kurzfassung

Eingebettete Systeme sind elektronische Komponenten, die meist in ein übergeordnetes System integriert und für eine spezielle Aufgabe optimiert sind. Entwurfsziele wie die maximalen Kosten, der maximale Leistungsverbrauch oder die minimale Performance resultieren aus dieser speziellen Aufgabe. Da diese Ziele bei jedem System anders sein können, werden spezielle Entwurfsverfahren benötigt, die den Entwickler unterstützen die besten Entwurfsentscheidungen zu treffen. Eine schwierige und kritische Entscheidung ist die Wahl der Systemarchitektur und die Zuordnung von funktionalen Blöcken zu Verarbeitungseinheiten. Um diese Entscheidung optimal treffen zu können, wurden in den letzten Jahren Verfahren für die sogenannte Exploration des Entwurfsraums entwickelt. Diese Verfahren ermöglichen das Evaluieren und Vergleichen verschiedener Architekturen und Zuordnungen.

Bestehende Werkzeuge, die den Entwickler bei diesem Schritt unterstützen, zielen in erster Linie auf den Entwurf eines Ein-Chip-Systems ab. Für eingebettete Systeme mit geringer Stückzahl rentiert sich ein Chipentwurf jedoch nicht, daher wird die geforderte Funktionalität typischerweise mit Standardkomponenten realisiert. Zusätzlich haben die meisten Verfahren Modellierungseinschränkungen: so werden zum Beispiel von kaum einem Werkzeug komplexe Datenstrukturen unterstützt, was die Höhe der Abstraktionsebene deutlich beschränkt.

In dieser Arbeit wird ein neuer Ansatz zur Exploration des Entwurfsraumes vorgestellt, bei dem Berechnungen, Kommunikation und Datenstrukturen getrennt modelliert werden. Diese Trennung erleichtert das Erzeugen verschiedener Hardware/Software Implementierungen und ermöglicht zusätzlich die Unterstützung von komplexen Datenstrukturen. Diese werden mittels optimierter Komponenten aus einer Bibliothek auf die jeweilige Zielarchitektur abgebildet. Das präsentierte Konzept zielt speziell auf den Entwurf von eingebetteten Systemen mit geringer Stückzahl ab. Es werden Methoden zur Verfügung gestellt, die das effiziente Abbilden von abstrakten Systemmodellen auf gängige Standardkomponenten unterstützen. Die Evaluierung verschiedener Hardware/Software Realisierungen eines Systems wird mit dem vorgestellten Modellierungsansatz deutlich vereinfacht. Es hilft dem Entwickler die beste Zuordnung der funktionalen Blöcke zur Systemarchitektur zu finden. Der Ansatz wurde anhand einer Fallstudie, bei der verschiedene Realisierungsmöglichkeiten eines "Voice over IP" Systems verglichen wurden, evaluiert. Im durchgeführten Vergleich mit einem traditionellen Entwurfsablauf ergab die Verwendung des neuen Konzeptes einen deutlich reduzierten Modellierungsaufwand.

Acknowledgements

This thesis and the underlying research results have been achieved as part of the VirtualVoIP project at the Institute of Computer Technology in cooperation with the company Frequentis AG. Without their initiative and their financial support this work would not have been possible. Particularly I want to thank Dr. Hans-Dieter Wohlmuth who supervised this work at Frequentis. He continuously supported me and helped me with enthusiasm to keep focused. In the meantime I became part of his hardware design team at Frequentis. Therefore, I look forward to our ongoing collaboration.
At this point I would like to express my sincere appreciation to Univ.Prof. Dr. Christoph Grimm and Univ.Prof. Dr.rer.nat. Ulrich Schmid for supervising this work and for their valuable feedback. At the Institute of Computer Technology I owe my gratitude to the whole Embedded Systems group for the great working environment. I am indebted to Dr. Jan Haase, the project head of the VirtualVoIP project, who always had a sympathetic ear for my problems. Especially I want to thank my colleagues DI (FH) Florian Schupfer und Jiong Ou, MSc for many fruitful discussions during the daily coffee break. Both became friends in the last four years.
I also want to thank all my friends, who helped me with fun and the required distraction in my spare time to gather energy for continuing my research. Last but not least, my parents and my partner Michaela receive my deepest thanks. They had always believed in me and encouraged me in stressful times.

Table of Contents

Abbreviations

ADC	Analog-to-Digital Converter
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
CDFG	Control Data Flow Graph
COTS	Components-off-the-Shelf
CPU	Central Processing Unit
CTL	Codesign Template Library
DAC	Digital-to-Analog Converter
DC	Direct Current
DSE	Design Space Exploration
DSP	Digital Signal Processor
eLOC	Effective Lines of Code
EMIF	External Memory Interface
ESL	Electronic System Level
FFT	Fast Fourier Transform
FIFO	First In - First Out
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FSMD	Finite State Machine with Datapath
GPIO	General Purpose Input/Output
GPP	General Purpose Processor
HDL	Hardware Description Language
HLS	High Level Synthesis
HW	Hardware
IP	Intellectual Property
IP	Internet Protocol
ISS	Instruction Set Simulator
KPN	Kahn Process Network
LIFO	Last In - First Out
LUT	Look-Up Table
MAC	Media Access Control
MAC	Multiplier-Accumulator
MoC	Model of Computation
MPSoC	Multi Processor SoC
NCO	Numerical Controlled Oscillator
OSI	Open System Interconnection
PC	Personal Computer
PCB	Printed Circuit Board
PCM	Pulse-Code-Modulation
PHY	Physical Layer Chip
QoS	Quality of Service
RAM	Random-Access Memory
ROM	Read-Only Memory
RTCP	RTP Control Protocol
RTL	Register Transfer Level
RTOS	Real-Time Operating System
RTP	Real-Time Transport Protocol
SFDR	Spurious Free Dynamic Range
SINAD	Signal-to-Noise and Distortion Ratio
SIP	Session Initiation Protocol
SLDL	System Level Design Language
SoC	System on Chip
SPP	Special Purpose Processor
SSRC	Synchronization Source Identifier
STL	Standard Template Library
SW	Software
TCP	Transmission Control Protocol
TDA	Tripartite Design Approach
TLM	Transaction Level Model
UDP	User Datagram Protocol
VHDL	Very High Speed Integrated Circuit HDL
VoIP	Voice over IP
XMPP	Extensible Messaging and Presence Protocol

1 Introduction

Ubiquitous computing and ambient intelligence are just two technological paradigms, which illustrate the evolution of computer technologies. Sensors and actuators as applied in building automation or the increasing number of intelligent devices used at home, show the pervasion of electronic devices in today's life. Such systems, so-called embedded systems, impose special demands on their designers. Design constraints heavily influence the design process and dictate the direction of optimization. The importance of design metrics like development and production costs, power consumption and performance may vary substantially for each design implementation. To keep pace with these challenges, new design methodologies have arisen. In recent years, the focus for hardware/software systems clearly moved to the so-called electronic system level. However, existing solutions have many drawbacks which reduce their applicability. In the following, the motivation for this work is presented. Thereby, an important factor are the disadvantages of current solutions especially regarding low volume embedded systems. The contributions and an outline of this work are also shown.

1.1 Motivation

This Section illustrates the importance of embedded systems and the importance of design space exploration for the design of such systems. Furthermore, requirements for an electronic system level tool, which supports the design space exploration of low volume systems, are derived.

1.1.1 Embedded Systems

In principle digital systems can be divided into two categories: general purpose and application specific systems. General purpose systems like personal computers are not designed for a particular application. They are rather optimized to provide a platform, which can perform a large variety of operations using application specific software. In contrast to this, application specific systems are designed and optimized for a dedicated application. These systems are typically embedded in a larger system and therefore commonly called embedded systems [Gup93, p. 1].

The worldwide market for embedded technology has more than doubled in the last ten years and was $ 113 billion in 2010. According to [43] it is expected to grow with a rate of 7 % per year to a value of around $ 159 billion in 2015. Fig. 1.1 shows the embedded products market revenue in 1998 and 2010 and its expected revenue for 2015. The significant revenue increase illustrates the importance of embedded systems. Today, embedded systems are almost everywhere, from a simple dishwasher to a complex home entertainment system. They are utilized in several different industrial sectors such as consumer electronics, telecommunications, automotive, avionics, medical and industrial automation [45].

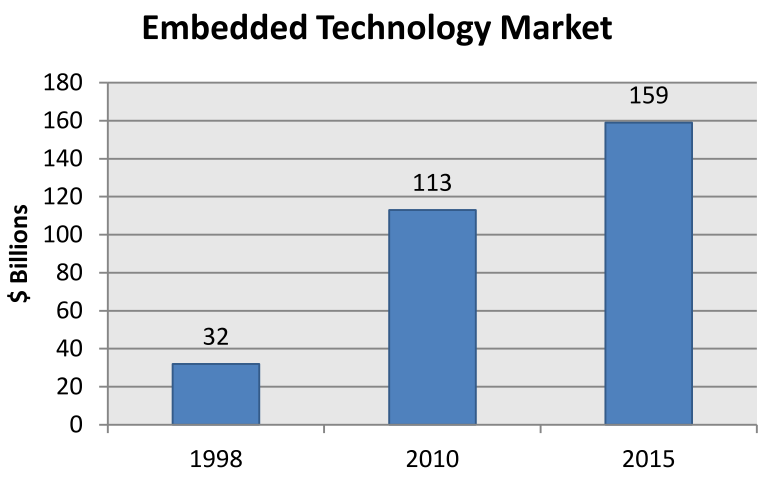


Figure 1.1: Worldwide embedded technology market 1998, 2010 and expected value for 2015 [43]. The significant revenue increase illustrates the importance of embedded systems.

As different as the application domains are, so are the requirements the systems have to fulfill. A consumer product typically has to have a high performance, a small price and optimally a small size. In contrast, for automotive electronics like an anti-lock braking system (ABS) safety, reliability and a long life cycle are important. Such design constraints heavily influence the design process and the optimization direction. Although conventional design approaches can be used, they are usually not sufficient to find the best solution. These challenges have led to new design methodologies [Koo96].

1.1.2 Electronic System Level

A conventional design approach separates the system specification into parts which are realized in hardware and parts which are realized in software. This early partitioning decision is typically based solely on the designers experience, which leads to suboptimal design solutions. To overcome this problem, design methodologies have been developed which focus on the design of the system as a whole rather than separating it into hardware and software design. First developments in this direction where HW/SW codesign approaches like Ptolemy [EJL$^+$03] or Polis [BCG$^+$97].

The continuation of this concept resulted in a switch of design effort to the so-called electronic system level (ESL). In [GB07] ESL is defined as "the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner." Hence, ESL design includes concepts and methodologies to get to know the system at an early design stage to make the best design decisions. A widely used approach is based on modeling and simulation. Specific system level design languages are utilized to model basic components of a system at a high abstraction level. The simulation of these components and their interaction provides further information concerning the system's behavior. Among others, this helps the designer to identify which components are better suited for hardware and which for software. Especially for designing systems on a chip (SoC), C-based languages like SystemC [OSC] gained acceptance [GAGS09, p. 327].

SystemC is basically a standardized C/C++ class library with facilities to model hardware [GLMS02, p. 11]. Comparable to hardware description languages like VHDL or Verilog, this includes possibilities to model hierarchy, timing and parallelism. A key advantage is the possibility to model systems at different abstraction levels. This enables the successive refinement of an abstract system model towards a concrete implementation. Another advantage of SystemC is the availability of high level synthesis tools, which automatically translate models at higher levels of abstraction to register transfer level (RTL) hardware implementations [BHS09]. This link from high level models to actual implementations is an important requirement for the further acceptance of electronic system level design [GAGS09, p. 2].

1.1.3 Design Space Exploration

The objective of electronic system level design is to increase the designers comprehension about a system. Thereby, making critical design decisions like the HW/SW partitioning is simplified. The systematic analysis of different solutions to such a design decision is called design space exploration, which today is an important part of system level design. The term results from the multi-dimensional space which is spanned by crucial design metrics such as power, performance or costs. Each possible realization is classified in the design space according to its design parameters. The space of valid design solutions is limited by given design constraints [Ham09, p. 23].

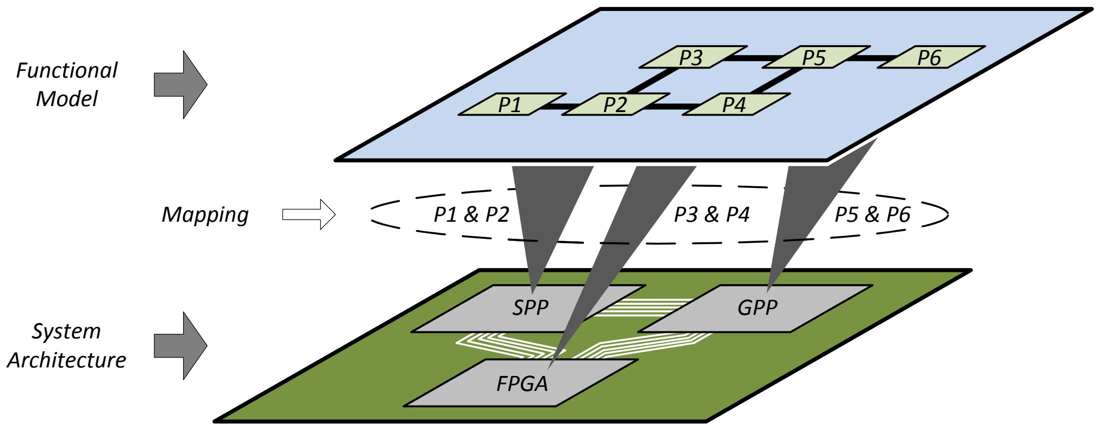


Figure 1.2: During design space exploration different application-to-architecture mappings are compared. One such mapping corresponds to the assignment of the high level model's processes to processing units.

The exploration of different solutions to a design decision can be performed at different stages of a design flow. A critical decision during system level design is the choice of an appropriate architecture and the mapping of functional components to this architecture. Fig. 1.2 illustrates this mapping. The typically pure functional high level model implemented using a system level design language consists of several communicating processes. These processes are mapped to an architecture consisting of different processing units like central processing units (CPU) or custom hardware blocks connected via communication interfaces [GAGS09, p. 124]. The architecture shown in Fig. 1.2 consists of a field-programmable gate array (FPGA), a special purpose processor (SPP) and a general purpose processor (GPP).

Several architectures with different processing units and communication systems are possible. The actual architecture also depends on the kind of realization. SoCs often use bus systems to interconnect several CPU cores and custom hardware blocks. If a board-level realization with standard components is chosen, the architecture is typically built out of general and special purpose processors. Custom hardware blocks are then realized using FPGA. These components are often interconnected with one-to-one communication interfaces.

Generating and evaluating different application-to-architecture mappings is a time-consuming process. Electronic system level tools support the designer performing this task. The aim of these tools is to simplify the two basic steps of the design space exploration (DSE): exploration and evaluation [KAL11, p. 23]. Starting with an abstract, functional system model, the designer should be supported to easily generate different application-to-architecture mappings. To find the best solution, each mapping has to be evaluated regarding its design metrics. Therefore, design parameters like costs, performance or power have to be estimated.

Tools which focus on design space exploration can be divided into two categories. On the one hand high level synthesis tools are promoted by their vendors as ESL solutions which support design space exploration. They can be used to generate different RTL implementations out of one and the same abstract system model. However, they mainly focus on hardware design and are not suited to explore different application-to-architecture mappings [GAGS09, p. 294]. On the other hand ESL solutions exist, which focus on the design space exploration of entire hardware/software systems. Many of these tools are still part of academic research and have various restrictions and limitations.

1.1.4 Problem Description

Existing ESL solutions have several disadvantages, which reduce their applicability for design space exploration. In this Section significant requirements for an ESL tool, which supports the designer during the mapping process, are derived. The focus is especially on low volume systems. Although their design flow is comparable to the design flow of SoCs, there are significant differences. Therefore, several general and particular requirements for an ESL tool for design space exploration are derived.

High Level of Abstraction

The architecture mapping is typically performed using a high level, functional system model. Since this model represents the pure functionality of the system without including details of the implementation, it has to have a high level of abstraction. Often algorithm implementations using high level language constructs are directly used in this model. To simplify design space exploration, ESL solutions have to support high level language constructs. The designer has to be able to directly use the functional model for DSE. Current ESL and high level synthesis (HLS) tools have certain restrictions, which limit the level of abstraction. Examples are complex data structures which are not supported by many current solutions. Therefore a requirement for an ESL tool is a high level of abstraction.

Link to Implementation

As already mentioned, this work focuses especially on embedded systems with low to medium quantities. A low expected production volume increases the importance of development costs. Thus, the realization of such systems as an SoC is not profitable. Rather components off-the-shelf (COTS) like FPGAs and general or special purpose processors are used. Such board level systems demand particular requirements from a design space exploration tool.

One important requirement is a link from system level design to the implementation to enable early prototyping. SoC designs are massively simulated before they are produced, since an error found after production may be expensive. However, this is different for board-level systems. In-system debug mechanisms like JTAG led to earlier prototyping. A lot of the test and verification effort is performed on existing prototypes.

A link to the implementation has further advantages not only for the design of low volume systems. If the used model can simply be translated or refined to an actual hardware or software implementation, low level tools can be utilized to more accurately estimate different design metrics. This allows a simpler and more accurate classification of different design solutions.

Full Target Support

Another important requirement is the support of the target architecture. This is basically important for all designs. Especially FPGA design denotes the mapping of an implementation to available hardware resources. Only if all available features of a target architecture can be exploited, it is possible to generate efficient prototypes which are close to a real implementation. Many of the current solutions focus on SoC design and therefore do not fully support the efficient mapping to FPGAs.

Application Domain Independence

Typically, a system can be divided into a control flow dominated part and a data flow dominated part. The control flow dominated part is usually realized as software on a general purpose processor. The more difficult design decision is the mapping of the data flow dominated part. On the one hand a typical target is an FPGA, which enables a massively parallel realization with a high throughput. On the other hand, a cheaper alternative is the realization using special purpose processors optimized for data flow dominated tasks like digital signal processors.

The aim of this work is to provide an ESL environment for architecture mapping, whereby the focus is especially on the FPGA/SPP partitioning decision. A limitation to a certain type of system like pure data flow systems can simplify tool development. However, modern systems are typically heterogeneous. Data flow dominated systems often also consist of timing and synchronization components, which cannot be modeled using pure data flow approaches. Therefore, another requirement for an ESL solution is the independence of the application domain.

1.2 Outline and Contributions of this Thesis

In the previous Section, four requirements for an ESL tool supporting a designer during design space exploration of a low volume embedded system have been derived. The approach presented in this work provides an ESL solution which fulfills all of these requirements.

The core component is the Tripartite Design Approach (TDA), which separates communication, computation and data structures at the system level. This separation allows the realization independent design of computation components. Hence, they can be directly synthesized and compiled at the same time without modifying the component's implementation. Combined with a library based approach for communication components and data structures, this provides a solid link from the system level model to actual hardware/software implementations. This approach simplifies the generation and evaluation of different architecture mappings. Realization independent design of computation components and libraries for communication components and data structures allow a rapid generation of prototypes and the utilization of low level tools to accurately estimate design parameters.

A library based approach further offers great flexibility. On the one hand predesigned standard components automate the switch from the high level model to the implementation. On the other hand application specific components can easily be integrated during the design process. Although the TDA approach focuses on data flow dominated designs, it is not restricted to them.

Another component of the presented approach is the Codesign Template Library (CTL). This is a data structure library which provides data structure implementations for the high level model and for hardware and software implementation. In contrast to existing approaches, complex data structures can be used at the system level. For hardware or software refinement, the high implementation is replaced by optimized, domain specific implementations. The possibility to use complex data structures at the system level significantly increases the level of abstraction.

The hardware implementations of the CTL have been optimized for mapping to FPGAs. The library user can influence the actual data structure to memory structure mapping. Thereby, available FPGA memory structures can be utilized efficiently.

This work is based on the following publications:

- P. Brunmayr, J. Haase, and F. Schupfer. Late Hardware/Software Partitioning by using SystemC Functional Models. In: Proceedings of the 3rd Asia International Conference on Modelling and Simulation (AMS 2009), pages 194-199, May 2009.
- P. Brunmayr, H.D. Wohlmuth, and J. Haase. An Efficient FPGA Implementation of an Arbitrary Sampling Rate Converter for VoIP. In: Austrochip 2009, pages 33-38, October 2009.
- P. Brunmayr, J. Haase, and C. Grimm. A Tripartite System Level Design Approach for Design Space Exploration. In: Proceedings of the 2010 Forum on specification & Design Languages, pages 50-55, September 2010.
- P. Brunmayr, J. Haase, and C. Grimm. A Hardware/Software Codesign Template Library for Design Space Exploration. In: Proceedings of the 2011 Electronic System Level Synthesis Conference, pages 5-10, June 2011.

A short outline of the thesis is given in the following:

Ch. 1 illustrates the importance of embedded systems and briefly presents the evolution of electronic system level design. The exploration of the design space is identified as an important tool to improve the quality of crucial design decisions. Additionally, requirements for an ESL solution particularly for low volume embedded systems are derived.

Ch. 2 describes the state of the art of the design of embedded systems. The main focus lies on C-based approaches and systems involving FPGAs. This includes system level modeling, high level synthesis as well as today's design methodologies used to design data flow dominated systems including FPGAs and digital signal processors (DSP). Additionally, related work focusing on design space exploration and hardware/software codesign is introduced. These works can be categorized into high level synthesis tools and so-called electronic system level tools. Academic as well as commercial solutions in both categories are presented and analyzed regarding their applicability for design space exploration of FPGA/DSP systems.

Ch. 4 presents the new Tripartite Design Approach (TDA). First the design of a realization independent computation module is derived. This so-called HWSW-Module forms the core component of the new approach. It is directly synthesizable and compilable at the same time. Then the Codesign Template Library is introduced. This library provides complex data structure with exchangeable implementations optimized for high level simulation, hardware and software implementation. Finally, the application of the new approach for system design is illustrated.

In Ch. 5 the TDA including the CTL library is applied to design an embedded voice over IP (VoIP) engine optimized for safety critical application areas. For the data flow dominated part of the engine, a design space exploration is performed to find the hardware/software partitioning which best fits the given design constraints. Finally, the effort for performing this design space exploration is measured and compared to a traditional approach.

In Ch. 6 the results of the case study are critically reflected. The achievement of the identified requirements for an ESL solution for board level systems stated in Ch. 1 is analyzed.

Finally in Ch. 7, the results and contributions of this work are summarized and a short outlook on possible future work is given.

2 State of the Art

In Ch. 1 design space exploration has been identified as an essential system level tool, which helps the designer to find the best application-to-architecture mapping. Since current solutions mainly focus on SoC designs, the aim of this work is the development of an approach supporting board level systems as utilized for low volume embedded systems. In this Chapter relevant state of the art design methodologies are presented. Of particular interest are thereby design phases directly related to the application-to-architecture mapping. The basis for this mapping forms a system level model derived from the specification. Therefore, Sec. 2.1 presents system level modeling basics. A demanded characteristic is the link to the actual implementation. For hardware design this link is formed by high level synthesis. Sec. 2.2 shows its basic functionality and analyzes the requirements a synthesizable model has to fulfill. Finally, in Sec. 2.3 implementation methodologies for architectures utilized by low volume embedded systems are discussed.

2.1 System Level Modeling

The first system model is often called an executable specification [GLMS02, p. 7]. It usually has a high abstraction level and is a direct translation of the specification into a system level design language (SLDL). The term executable indicates the possibility to simulate the model which is not possible with a textual specification. Typically, it is a pure functional model completely independent of any intended implementation. This model is used throughout the design process as reference model. In the context of system level modeling the subjects model of computation and separation of communication and computation have to be considered. They are presented in Sec. 2.1.1 and Sec. 2.1.2 respectively.

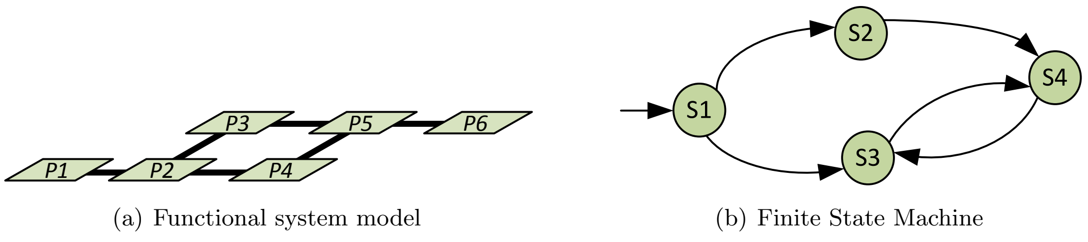
2.1.1 Model of Computation

According to Gajski a model of computation (MoC) is "a generalized way of describing system behavior in an abstract, conceptual form" [GAGS09, p. 50]. The MoC builds a formal basis for the designer to model a system. It defines components of a system, the organization of computation in those components and communication between them. Several MoCs exist and they differ in their provided features, complexity and expressiveness. Depending on these characteristics, they are used for different types of applications and in different phases of the design process [Mar10, p. 28].

A simple example of an MoC is the imperative model of computation, which is realized by sequential programming languages like C/C++. The imperative model describes the behavior of a system as a sequence of consecutive instructions. It forms the most common MoC for software implementation. Although, modern programming languages offer facilities like multi threaded programming and object oriented design, it is most suitable for modeling pure untimed functionality of single processing units [GAGS09, p. 51].

For digital hardware development its counterpart is the discrete event MoC. It is realized by hardware description languages like Verilog or VHDL. The discrete event MoC describes the system as an ordered sequence of events. Whereby, an event corresponds to a state change at a certain instant in simulation time. It is perfectly suited to model concurrency and therefore to model parallelism as it is required for digital hardware design [GAGS09, p. 171].

Both MoCs are implementation domain specific. They have been developed for a particular use case, namely the development of software and hardware respectively. System level models are usually implemented utilizing high level, realization independent MoCs. While the imperative and the discrete event MoC are implementation domain specific, high level MoCs are typically application specific. They can be broadly separated into two categories: process based and state based MoCs [GAGS09, p. 52].



(a) Functional system model (b) Finite State Machine

Figure 2.1: High level models of computation can be broadly separated into process based and state based MoCs. A functional system model is an example for a process based model of computation. The finite state machine is the simplest example for state based models of computation.

Process based MoCs are usually applied to the design of data flow applications. They model the system as several concurrent processes. Each process internally models computation using a sequential programming model. For the communication among the processes point-to-point channels are utilized. An example for a process based MoC is the Kahn Process Network [Kah74]. It utilizes infinite first in first out (FIFO) data structures as communication elements. Process based models are untimed models which are applicable to model the pure functionality of a system at a high abstraction level. The functional model already illustrated in Sec. 1.1.3 is a simple example for such a process based model. It is shown again in Fig. 2.1(a). In contrast, state based models are applied to design control dominated applications. They model the system in terms of states and transitions between states. The simplest example for a state based MoC is the Finite State Machine (FSM). Fig. 2.1(b) shows a simple FSM with four states.

The first system model is usually implemented using a SLDL. As already mentioned, the most common SLDLs are C-base languages like SystemC. As classic hardware description languages (HDL), their basic MoC is the discrete event model. However, they provide facilities to model the system at a wide range of abstraction levels. Hence, different MoCs can be realized using one and the same design language [GAGS09, p. 327].

Particularly considering data flow dominated designs, the following design flow can be summarized. The design typically starts with a process based functional model as illustrated in Fig. 2.1(a). This model is implemented using a C-based SLDL. Throughout the development process, it is refined towards an actual implementation. During this refinement, the pure process based MoC often has to be given up because timing and synchronization issues become important. A pure untimed MoC is not capable of modeling such issues. Therefore, different MoCs are utilized throughout the design process.

2.1.2 Separation of Communication and Computation

Today, the separation of communication and computation is a well-established modeling paradigm for system level design. In general, the separation of different aspects of a design is called orthogonalization of concerns and has been exploited e.g. for designing digital hardware by separating functionality and timing. Its utilization for system level design has been presented by Keutzer *et al.* [KMN+00]. The idea is to separate parts of the design such as function and architecture or communication and computation to handle the ever increasing complexity of the design process.

A clear advantage of this separation is the possibility to reuse both communication and computation components at all levels of abstraction. It allows the separate refinement of communication and computation components, which is important for the successive refinement of system level models to an actual implementation.

Modern C-based SLDLs like SystemC implement the separation of communication and computation using modules, ports and channels. Computation is realized in processes, which are encapsulated in modules. Each module can consist of one or more processes, whereby different types of processes exist. The most common process type for high level models is the `SC_THREAD` which is comparable to a software thread. During simulation it is executed once and the execution can be interrupted using `wait()` statements. Thereby, it is possible to synchronize to an event or to wait a specified amount of simulation time. However, the typical high level module is implemented purely untimed [GLMS02, p. 25].

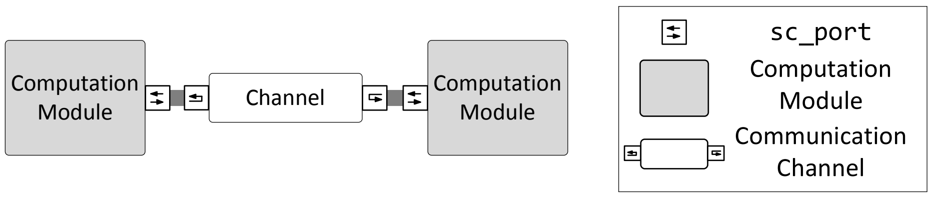
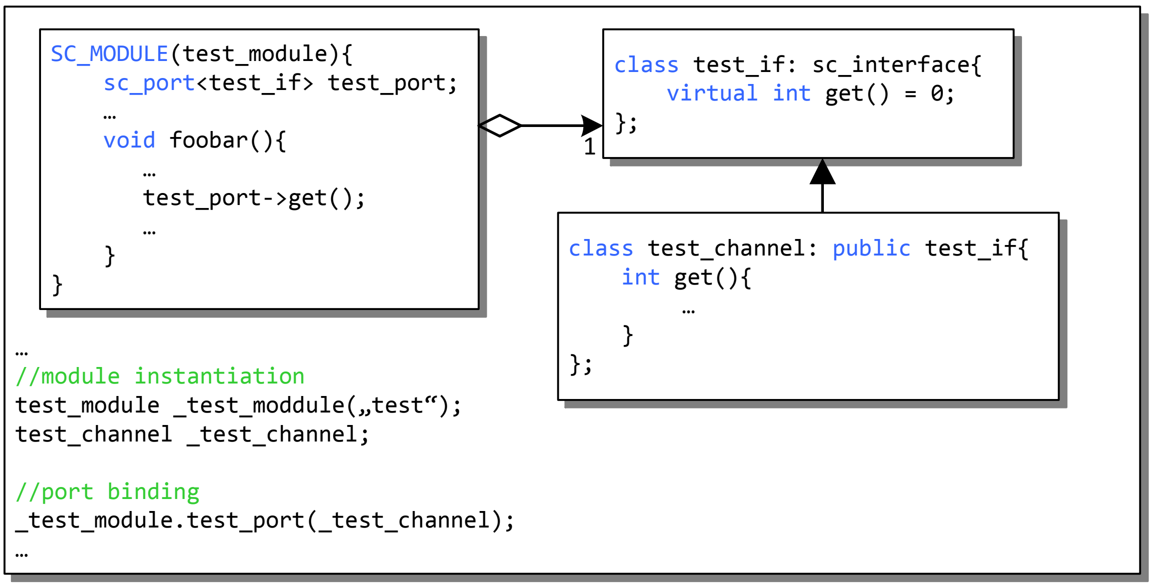


Figure 2.2: Two SystemC computation modules are connected via a communication channel. The separation of computation and communication is realized using an interface and the `sc_port` class.

Fig. 2.2 shows two computation modules. The communication among these modules is hidden in a communication channel. In a high level model a channel with an abstract form of communication is used. During system refinement, this channel can be replaced by channels with more concrete communication protocols. Following the orthogonalization of concerns concept, communication and computation are implemented in separated design components. They can be refined independently from each other [GLMS02, p. 153].

The actual connection between module and channel is realized using the `sc_port` class. A port provides a function interface to the processes in the module. Each function call to a port is rerouted to the connected channel which provides the actual implementation of those functions. At the instantiation of a port an interface has to be specified. This interface class defines the functions a connected channel has to implement.

This connection concept is based on dynamic polymorphism which is a key component of object oriented programming [Sch98, p. 426]. The interface consists of so-called virtual functions. The keyword virtual indicates a member function, whose implementation can be overridden in an inherited class. Different implementations can be provided by different inherited classes. It is possible to implement a class which operates on the interface via a pointer. To which implementation the pointer actually points is resolved during runtime [BD04, p. 129].



```
SC_MODULE(test_module){
    sc_port<test_if> test_port;
    …
    void foobar(){
       …
       test_port->get();
       …
    }
}
```

```
class test_if: sc_interface{
    virtual int get() = 0;
};
```

```
class test_channel: public test_if{
    int get(){
        …
    }
};
```

```
…
//module instantiation
test_module _test_moddule(„test“);
test_channel _test_channel;

//port binding
_test_module.test_port(_test_channel);
…
```

Figure 2.3: Example of connecting design components using dynamic polymorphism. The channel `test_channel` implements the interface defined by `test_if` and can thus be connected to the port `test_port`. This corresponds to the standard way of connecting channels to modules in SystemC. [BD04, p. 129]

The pointer to the interface is in SystemC hidden in the `sc_port` class and the actual interface class, which defines the available operations, can be set via a template parameter. In Fig. 2.3 a simple example of a port to channel connection is shown. An interface named `test_if` is defined with a member function `get()`, which returns an integer value. By adding "$= 0$", the function is defined pure virtual, which forces every inherited class to implement the defined function. Further, a concrete channel is named `test_channel`. It is inherited from the defined interface and it implements the `get()` function.

The `test_if` interface is used in a module named `test_module`. This module has an `sc_port` to which any class inherited from `test_if` can be connected. In the top level module, the module and the channel are instantiated and connected via port binding, which is done using the overloaded bracket operator. The connection between the module and the channel is resolved during runtime. In the presented example, this happens, when the port is accessed and the `get()` function is called in the `foobar()` function.

The connection of channels with modules via ports and interfaces enables the separation of communication and computation in SystemC. Both channels and modules can be exchanged independently of each. This enables the independent refinement of communication and computation.

2.2 High Level Synthesis

An important methodology to automate the refinement process from a system level model to an actual hardware implementation is the high level synthesis. The term high level or behavioral synthesis denotes the automatic generation of an RTL description from a hardware description on a higher level of abstraction. Different approaches use different design languages for this high level description. Today, C-based tools are most commonly used [GAGS09, p. 327]. Hence, the input description is designed using C/C++ or SystemC. In contrast to system level synthesis, HLS confines itself to the synthesis of single hardware blocks. However, such a hardware block might consist of different hierarchy levels with various parallel processes.

The principle functionality of high level synthesis is presented in Sec 2.2.1. The typical structure of an input description and limitations and particularities of modern synthesis tools concerning data types and structures are illustrated in Sec. 2.2.2 and Sec. 2.2.3.

2.2.1 Basic Functionality

Various different high level synthesis algorithms exist. In principle, all modern HLS tools have the basic flow shown in Fig. 2.4 [CMGT09]. As already mentioned, the synthesis process starts with a high level description of a hardware block. Further inputs are design constraints and an RTL component library. Examples for design constraints are resource and timing constraints such as a maximum latency or the amount of available hardware resources. The RTL component library consists of data path elements, which are available in the target architecture. In the end, the generated RTL implementation is built from these data path elements. Additionally, the library contains component characteristics like timing or area information.

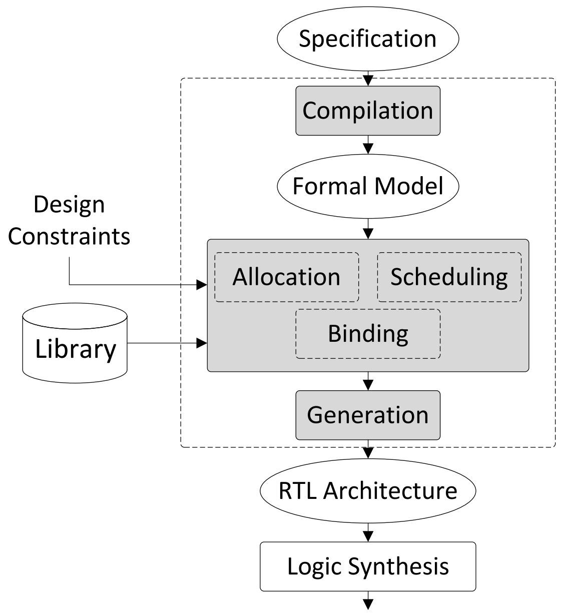


Figure 2.4: Typical high level synthesis design flow [CMGT09]. The key steps are allocation, scheduling and binding, which are controlled by design constraints.

The high level description, which is basically a functional specification, is compiled like a software program and translated to a formal model, typically a control data flow graph (CDFG). The compilation process is used to apply several design optimizations comparable to the optimizations of a software compiler, e.g. dead code elimination, constant folding and common sub-expression detection. The generated CDFG illustrates data and control dependencies of the different operations. The data dependencies further provide the natural parallelism of the computation, which corresponds to the maximum parallelism of the final hardware design.

After generating the CDFG, three key steps of the HLS are performed: allocation, scheduling and binding, see Fig. 2.4. During these steps, crucial design decisions are made. The first step mentioned, the allocation, denotes the selection of hardware resources. Thereby, for each operation type in the specification at least one hardware resource from the component library is chosen. Some tools generate optimized data path elements for specific parts of the computation during the synthesis process to further improve the overall design quality [49].

The second of the three key steps, the scheduling, denotes the assignment of operations to clock cycles. Thus, in this step it is defined which operations are performed in parallel and which sequentially in different clock cycles. The scheduling is influenced significantly by the given design constraints. A resource limitation of two multipliers e.g. forces the synthesis tool to distribute four multiplications of the specification among at least two clock cycles.

The third step is the binding. In this step, the operations are assigned to actual resources. Registers are instantiated for variables which hold a value across a clock cycle boundary. Variables which are solely used to exchange an intermediate result between two operations within a single clock cycle, are realized as simple connections. The order of the three mentioned steps allocation, scheduling and binding depends on the algorithms used. Sometimes all three operations are performed simultaneously to achieve better results.

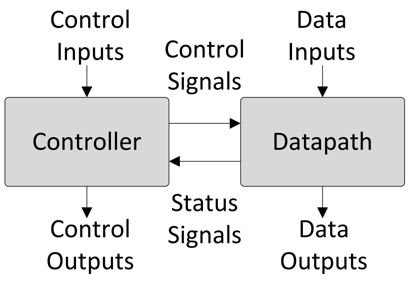


Figure 2.5: High-level block diagram of a typical RTL architecture [GAGS09, p. 207]. A finite state machine (controller) observes and controls the data path.

After making all critical design decisions during allocation, scheduling and binding, an RTL model can be generated. The resulting model is basically a finite state machine with datapath (FSMD). Fig. 2.5 shows an abstract block diagram of the RTL architecture consisting of a controller and a datapath. The controller reflects the state of the hardware block. It controls the datapath in each clock cycle and receives status signals from it. Additionally, the controller has inputs and outputs to interact with other components. The datapath itself reads in data, executes the operations defined in the functional specification and produces some results.

The resulting RTL design is typically implemented using Verilog or VHDL and can be directly processed by a logic synthesis tool. During logic synthesis the design at this point, basically consisting of combinatorial logic and registers is further optimized and translated to a so-called netlist. If it is synthesized for an FPGA, the design components are then placed and interconnected during the so-called place and route process typically performed by a tool provided by the FPGA vendor. Finally, a binary FPGA configuration file is generated [HD07, p. 151].

2.2.2 High Level Specification

As already mentioned, most HLS tools support an input specification in C/C++ or SystemC. Since the timing is generated by the tool using scheduling algorithms controlled by design constraints, it is possible to implement at least parts of the specification in an untimed manner. In the following, a typical synthesizable SystemC description is shown. It is also mentioned what the corresponding C/C++ description would look like. The information concerning the supported input specifications of modern HLS tools is taken primarily from the user guides of the following tools: Calypto Catapult C [Fin10], Cadence C2Silicon [44], Xilinx AutoESL [42] and ForteDS Cynthesizer [50].

```
SC_MODULE(Example) {
  sc_in<bool> iClk;
  sc_in<bool> iReset;
  sc_in<bool> iValid;
  sc_in<int>  iData;
  sc_out<int> oData;

  SC_CTOR(Example) {
    SC_CTHREAD(Foobar, iClk.pos());
    reset_signal_is(iReset, 1);
  }

  void Foobar() {
    //reset cycle
    ...
    wait();
    while(1) {
      //input protocol
      while(!iValid.read())
        wait();
      tmp = iData.read();

      //untimed algorithm
      ...

      //output protocol
      oData.write(tmp);
    }
  }
};
```

Listing 2.1: Basic structure of a synthesizable SystemC module. The characteristic structure consists of a cycle accurate input and output protocol. The actual computation is implemented untimed.

In Lst. 2.1 the code of simple, synthesizable SystemC module is shown. It contains all the essential components a synthesizable module has to have. In SystemC the functional unit, which is synthesized at once, is implemented using an SC_Module. The interface of the hardware block is specified via pin-accurate input and output ports. Since synchronous hardware is modeled, most HLS tools demand a clock and a reset input of the synthesizable module. The actual computation is implemented using an `SC_CTHREAD`, which is a special thread for behavioral hardware description. It is called a clocked thread, which means that timings in the thread can only be expressed in terms of clock cycles. Each call of the function `wait()` corresponds to a delay of one clock cycle. The corresponding clock signal is assigned during the thread declaration, see line 9 in Lst. 2.1, where `iClk` is assigned to the thread `Foobar`. The reset signal is assigned using the function `reset_signal_is()`. Since the occurrence of a reset restarts the thread execution, the code up to the first wait is called the reset cycle.

A synthesizable description of the same functional unit in C/C++ would be using a simple untimed C function. Input and output ports are defined using function parameters and it is usually not necessary to declare a reset and a clock port.

By scheduling the design, the HLS tool automatically decides when which operation is executed. This enables the generation of different optimized hardware implementations from one and the same untimed behavioral model. However, to ensure the proper interoperability with other hardware blocks, it has to be specified when inputs are read and when outputs are written. Very often it is required to fulfill a more complex I/O protocol. In SystemC, these protocols are usually cycle accurate specified. In the shown example, the design waits until the control input `iValid` is set to high, only then is the input read, see line 19 in Lst. 2.1. Therefore, high level SystemC specifications often have the typical structure shown in Lst. 2.1: input protocol, untimed algorithm and output protocol.

To support completely untimed specifications as well, HLS tools provide predefined I/O protocols which can be selected e.g. via design constraints. In pure C/C++ designs typically only such predefined I/O protocols are available. Some tools introduce non-standard keywords, which correspond to the `wait()` function in SystemC, to enable the description of user defined I/O protocols in C/C++ [44].

SystemC provides facilities to realize explicit parallelism and hierarchy. Thus it is possible to synthesize a complex system, by instantiating several submodules in a top level module. It is also possible to have different `SC_CTHREADs` in one module at the same hierarchy level. In C/C++ this is more difficult. Hierarchy is usually realized using subfunctions. Primarily, the synthesis tool has to recognize the parallelism in the sequential specification and it has to exploit it.

2.2.3 Data Types and Data Structures

There are several particularities concerning data types and data structures and their utilization in a synthesizable high level specification. Significant differences exist between their usage for software and hardware implementation. On the one hand this results in several limitations so that not each data type and data structure is synthesizable. On the other hand specific extensions are required to ensure an efficient mapping of data structures to memory structures.

Essentially, simple data types are not a problem. The same basis data types like integer or boolean are used for both hardware and software design. Hence, the same operations are supported. The main difference concerns the used bit width. Typically, software data types have bit width corresponding to the native width of the processor's data path. The native bit width is usually 8, 16 or 32 bit in high performance systems maybe 64 bit. A smaller bit width than the data path width does not lead to increased performance. Quite the opposite, it may result in lower performance. Therefore, data types with arbitrary bit width are usually not used for software design. This is different for hardware design. A reduced bit width results in reduced hardware resources and therefore in reduced costs. Due to that, SystemC supports integer and fixed point data types with arbitrary bit width. Pure C/C++ synthesis tools often introduce proprietary data types of this nature [Fin10], [42].

Not all data types are supported by high level synthesis tools, e.g. floating point data types like double or float are not supported. Also high level language features like classes, operator overloading and templates are not accepted by some tools.

The particularities concerning data structures result from the memory model used. A software designer does not have to care about the actual structure of the memory. Typically a theoretically infinite, linear memory with a single address space is expected. In contrast, hardware design requires more complex memory models. On the one hand data structures can be realized using registers. This allows the parallel access to all elements of the data structure. Further, it is possible to utilize dedicated memory blocks. Different data structure to memory block mappings are possible. One memory block may also have several read and write ports, which allows parallel access to some of the elements. Each memory block has its own address space. To efficiently utilize the available memory resources in hardware, some HLS tools require specific code constructs.

Via design constraints, simple arrays can be mapped either to memory blocks or to registers. Some tools also support the mapping of different arrays to one and the same memory structure [Fin10], [44]. For more complex mappings some tools require the explicit instantiation of specific memory classes, which are generated by the synthesis tool [50].

HLS tools particularly have restrictions concerning the utilization of pointers and references. Both constructs are based on memory addresses. Thus, code which uses these constructs explicitly expects an infinite linear memory model. Most synthesis tools support pointers and references only if they can be resolved at compile time. If this is not the case, the addresses have to be adapted for the segmented hardware memory model. Thereto, Semeria *et al.* presents a solution in [SSDM01]. In his work, the linear memory is divided into different segments. An address is then translated into a segment address and an index. The segment address denotes the memory block to which the pointer refers to, while the index addresses the memory cell in the block itself. In this way, it is possible to map the pointer concept to hardware.

Another software concept, which leads to restrictions if used for high level synthesis, is the dynamic memory management. Currently, no commercial HLS tool supports it. Therefore, complex data structures like the container classes of the standard template library cannot be synthesized. In [CGM$^+$09] an approach for synthesizing the C constructs `malloc` and `free` is presented. This solution only works if the maximum size of the heap can be calculated at compile time. If this is possible, the heap is realized as a static array. A kind of hardware memory allocator manages memory allocation and deallocates, when the commands `malloc` and `free` are called. The disadvantage of this solution is that the whole heap is realized as one memory block. This prevents the efficient mapping of different parts of the heap to different memory blocks.

This disadvantage does not apply to Semeria's solution. As already mentioned in connection with pointers, this solution divides the linear memory into several segments. For each segment, a hardware allocator is instantiated. Obviously, this leads to a significant amount of hardware resources, which are required to realize the memory allocators. The tool tries to recognize situations in which a static realization without memory allocator is possible. However, this is only recognized under certain circumstances.

As in software, the realization of dynamic memory management leads to a certain overhead. The mentioned approaches only work with pure C. There is currently no work, which synthesizes dynamic memory management used with C++. Further, none of these theoretic solutions has been integrated into a commercial HLS tool, which only support static memory management.

2.3 System Architectures for Low Volume Embedded Systems

As already mentioned, this work focuses mainly on embedded systems with low to medium quantities. For such systems, development costs are more important than production costs. Thus, they usually have architectures built from off-the-shelf components and custom hardware blocks are typically realized using configurable ICs like FPGAs.

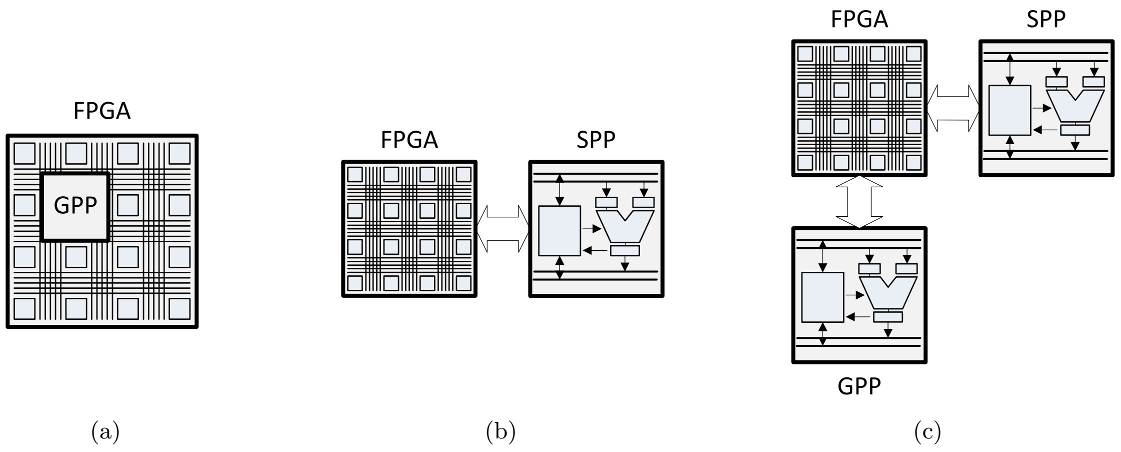


Figure 2.6: Typical system architectures with FPGAs. Depending on design constraints a single chip but also multi chip designs might be the best solution.

Fig. 2.6 shows example architectures involving FPGAs. The simplest HW/SW system consists solely of an FPGA. Using so-called soft or hard core processors it is possible to build a complete HW/SW system with just one FPGA, see Fig. 2.6(a). The difference between a soft and hard core processor is whether the core is realized using configurable logic or using a dedicated processor core, which is embedded in the FPGA. Examples for soft cores are the MicroBlaze from Xilinx [64] or the Nios II from Altera [39]. Hard core processors are integrated e.g. in the Xilinx Virtex 4 FPGAs [65] (PowerPC) or in the Actel SmartFusion [37] devices (ARM Cortex-M3). Using these embedded processors, a large variety of HW/SW partitionings, from a simple coprocessor to a complete hardware system, can be realized. However, special or general purpose processors are typically cheaper than FPGAs. Therefore, it can be useful to use a separate processor instead of a hard or soft core on the FPGA. The architecture FPGA and SPP as shown in Fig. 2.6(b) can for example be used for data flow dominated applications. In this case a digital signal processor is used as special purpose processor. More complex systems might require an architecture as shown in Fig. 2.6(c), where an FPGA is connected to an SPP and a GPP. Of course various combinations of these architectures are possible.

Architectures involving FPGAs and DSPs are of particular interest for the presented design approach. As already mentioned, the easier part of the partitioning is usually the identification of control flow dominated tasks, which can be realized on a GPP. The more difficult part is to decide whether to realize data flow dominated tasks on an FPGA or on an DSP. The massive parallel computation components on an FPGA can be utilized for very high performant implementations.

Further, hardware can be optimized application specifically. DSPs on the other side, are cheaper and designing them using C/C++ simplifies the design process compared to FPGAs. Today DSPs operate at a very high clock frequency, which leads to an acceptable performance as well. Therefore, FPGAs and DSPs are presented in more detail in the following Sections. Thereby, both the specifics of their architecture and the used design methodology is elaborated.

2.3.1 Field Programmable Gate Arrays

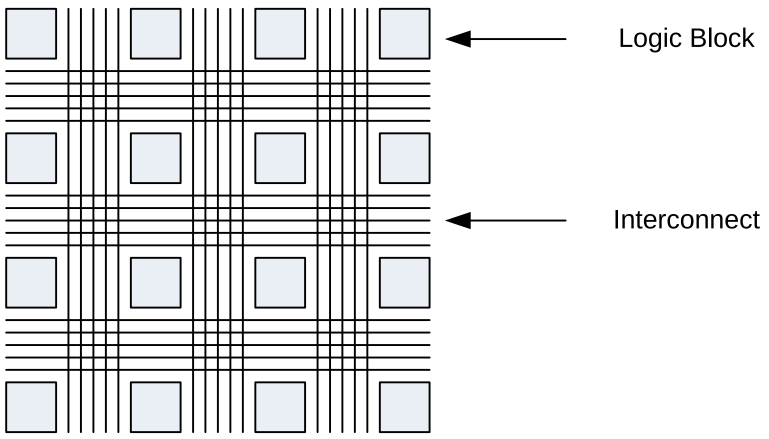


Figure 2.7: Principle structure of FPGAs [HD07, p. 7]. Programmable logic blocks are connected via the reconfigurable interconnect.

The principle structure of an FPGA is shown in Fig. 2.7. The basis are so called logic blocks, which provide facilities to implement logic functions. The logic blocks are connected by the reconfigurable interconnect [HD07, p. 7], which enables the routing of signals between logic blocks. A simple logic block is shown in Fig. 2.8. In principle it consists of a lookup table (LUT) usually built using static random-access memory (RAM). With this LUT, the logic function is realized. Additionally, a flip-flop is connected to the output of the LUT to build sequential logic. With a multiplexer at the output, either the combinatorial output of the LUT or the output of the flip-flop can be used for further processing. Of course the actual design of such a basic logic block differs from manufacturer to manufacturer. In Xilinx FPGAs, the smallest entity is called a slice and consists of two LUTs, two flip-flop, multiplexers and additional specialized logic like a carry logic for the realization of adders [66]. In Altera FPGAs a comparable entity is called adaptive logic block [38]. Modern FPGAs additionally have different other integrated hardware blocks. The usage of such specialized blocks enables a more efficient resource usage and faster designs than if everything is realized using configurable logic blocks. On the one hand blocks are integrated, which are heavily used, like memory blocks or multipliers. On the other hand more complex components for specific applications, like a Ethernet media access control (MAC) core or the already mentioned processor cores are integrated as well [66].

2.3.1.1 FPGA Design

Today FPGAs are mostly designed using hardware description languages like VHDL or Verilog. The modeling is typically done at the so-called register transfer level (RTL) [Rus11]. This abstraction level is characterized by the fact that registers, hence memory elements, are modeled

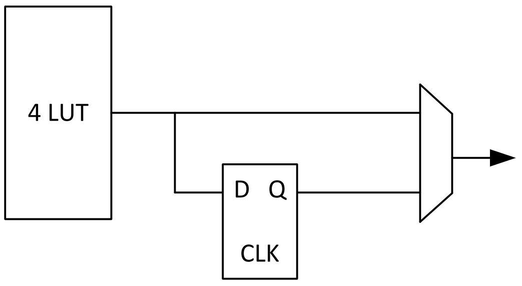


Figure 2.8: Basic structure of a logic block in an FPGA [HD07, p. 6]. The logic function is realized via a lookup table. The result can be used directly or stored in a flip-flop.

explicitly, while the logic inbetween is modeled abstractly and untimed. The position of the registers in the hardware circuit is defined by the designer. This influences the amount of logic a signal has to pass in one clock cycle. Thereby it further affects the maximum frequency, with which the circuit can be clocked. The logic synthesis tool optimizes the logic and maps it to the hardware components available in the FPGA. Most of the logic is realized using LUTs. However, as already mentioned, the FPGAs often have embedded components like multipliers or adders, which are utilized by the logic synthesis tool automatically. The modeled registers are mapped to the flip-flops in the logic blocks.

```
module RAM_512X16_1(DIN, RW, ADR, DOUT, CLK);
      input [15:0] DIN;
      input RW;
      input [8:0] ADR;
      input CLK;
      output [15:0] DOUT;
      reg [15:0] DOUT;
      reg[15:0] mem[511:0];

         always @(posedge CLK)
          begin :thread1
             if (RW) begin
                mem[ADR] = DIN;
             end
             else begin
                DOUT <= mem[ADR];
             end
         end
endmodule
```

Listing 2.2: Verilog template for dual port memory with 512 cells each 16 bit.

To use embedded memory blocks of an FPGA, HDL templates, like in Lst. 2.2, can be used. An other alternative is the explicit instantiation via the components name. However, this makes the HDL code technology dependent. Lst. 2.2 shows the Verilog template of a dual port memory. The example generates a block memory with 512 memory cells each with 16 bits. Typically a block RAM is used to realize this memory.

For the realization of logic, the designer does not have to care about the actual FPGA structure. The synthesis tool will find a proper mapping to the available resources. However, this is different for block RAMs. It is important how the designer utilizes the available components. Two medium sized data structures for example can be mapped to one block RAM because of the two read/write ports. The synthesis tool is not able to do such optimization by itself. If the code template for a block RAM is used twice, two block RAM resources are always used. The responsibility to efficiently use the memory blocks on the FPGA lies with the designer.

If high level synthesis is used, the RTL structure is automatically generated by the HLS tool. To ensure an efficient use of block RAM resources, the HLS tool has to choose an efficient mapping by itself or it allows the user to control this mapping process. Most tools leave the mapping decision to the designer and enable the mapping of arrays to block RAMs. For more complex mappings, some tools require the instantiation of special memory classes [50]. Obviously, this leads to a tool dependent source code.

2.3.1.2 In-System Debugging

A large portion of the design process is typically spent for simulating and testing the design. For application-specific integrated circuits (ASIC) exhaustive testing up to a coverage of almost 100 % is inevitable. The reason are the immense cost of a redesign after production. Whereas the effort for reprogramming an FPGA is negligible. Further, it is more complex to find the cause of an error in an ASIC during operation. Test logic has to be added explicitly. FPGAs by default have a JTAG [oEI03] interface via which it is configured. This interface can easily be utilized for in system debugging [Sim10, p. 133]. FPGA manufacturers provide the required software to integrate a kind of a logic analyzer into the design, which can trace certain signals during operation [41, 69]. Thereby, it is simplified to find design and integration errors. It can be very time consuming and computationally intensive to reconstruct specific real life situations in simulation. Due to that reason, FPGA designs are simulated less and the employment of a prototype for in system debugging is performed earlier in the design process.

2.3.1.3 Design Parameter Estimation

Another important point for designing systems including FPGAs is the estimation of design parameters to evaluate the fulfillment of design constraints. Interesting design parameters for FPGAs are e.g. the power consumption, the performance and the required hardware resources, which influences the required FPGA size and therefore the costs. If high level synthesis is used, the RTL design is generated by the HLS tool according to design constraints set by the user. Typically, it is possible to define the clock frequency and for example the desired latency. After synthesis, the designer gets information about the actual latency and an estimate of the required hardware resources. More accurate estimates of the required hardware resources are provided in later design steps, e.g. after logic synthesis or after place and route.

More difficult is the estimation of power. There are different approaches to estimate power at a higher abstraction level [AFMS07]. However, the lower the abstraction level, the more accurate is the estimation of the power consumption. FPGA manufacturers provide tools for power estimation, which analyze the netlist [67, 40]. The actual input data sequence significantly influences the amount of switching in the circuit and as a consequence the power consumption. Therefore, analysis tools take switching information generated via simulation into consideration. Whereby, more realistic simulation data, leads to a more accurate power estimation.

2.3.2 Digital Signal Processors

The alternative for realizing high performance data flow dominated applications is a digital signal processor. A DSP is a microprocessor, whose architecture is specialized and optimized for signal processing operations. The first commercially successful DSP, which had all major components that are today characteristic for a DSP, was the TMS32010 from TI [San09]. One of these characteristic components is a multiply-accumulate (MAC) engine, which consists of a multiplier followed by an accumulator. This kind of operation is very often needed in signal processing e.g. a digital finite impulse response filter can be realized solely using such operations.

Further DSPs are typically not based on the conventional Von-Neumann architecture, which uses a single bus to connect the memory, in which both data and instructions are stored. Instead the Harvard memory architecture with a separated data and instruction memory connected via two independent buses is used. This increases execution speed, since both data and code can be loaded concurrently [Noe05, p. 147].

These are only two typical characteristics of DSPs. In general, modern signal processors have various different optimizations to execute data flow dominated tasks more efficiently. An example for such an optimization are built in hardware structures, which allow a fast execution of loops [Keh05]. To exploit these architecture optimizations, DSPs have been programmed predominantly using assembly language. However this has changed in recent years. Currently, all major DSP vendors provide C/C++ compilers, which are able to exploit the specific architecture optimizations, see TI [22], Freescale [11], Analog Devices [1]. These compilers generate optimized implementations, so that a manual refinement using assembly language is not necessary anymore. Further, the shortening of the so-called time to market forces designers to switch to development methodologies on a higher level of abstraction. Today, inline assembler is used only in rare cases, e.g. for implementing very time critical functions.

Originally, DSPs were merely used as coprocessors. However, the complexity of the application, which is realized on a DSP has increased steadily. To keep track with the raising complexity, realtime operating systems are used more and more. This allows the separation of the application into several parallel tasks. Freescale and TI both provide their own real-time operating system (RTOS) for their DSPs, see [58] and [18]. Like common RTOS, they have facilities for interrupt handling and inter process communication.

As already mentioned, an important point in designing embedded systems is the adherence to design constraints. Therefore design parameters like performance, power and cost have to be measured or estimated. To make a clear statement whether a particular DSP is appropriate for an application, e.g. the execution time of a certain algorithm has to be evaluated. Such estimations influence the choice of the DSP and further influence the overall system costs. Methodologies for the estimation of performance are available on different abstraction levels. The exact execution time can be evaluated by executing the compiled source code on an instruction set simulator (ISS) or directly on the DSP. Typically DSP vendors provide ISS integrated into their development environment, see [51] and [47]. Thereby, it is possible to simulate solely the DSP core or the whole chip including all peripherals.

Similar methods are provided to estimate the power consumption of a DSP running an application. On the one hand, different approaches exist which try to estimate the power consumption already at the system level. On the other, DSP vendors provide more accurate low level methodologies for this estimation, e.g. TI offers estimation spreadsheets for its DSPs [59]. These spreadsheets calculate an estimation of the power consumption based on the used clock frequency and voltage and, via estimates of the utilization of the CPU and the peripherals. Analog Devices does not provide complete spreadsheets, however, it provides application notes, which help designers to estimate the power consumption for a certain application [46].

Both the utilization of compilers for high level languages like C/C++ and the usage of RTOS raise the level of abstraction for the designer. Thereby, a relatively simple mapping of different complex, parallel algorithms to a DSP is enabled. Although time critical important parts are still optimized using assembly language, these facilities simplify the overall development process especially in early design phases like system level design. Concurrently, system level design decisions require the utilization of low level estimation tools to get accurate design parameter estimates.

2.4 Summary and Evaluation

In the previous Sections different state of the art design and implementation methodologies around system level design space exploration have been presented. The basis for the exploration task usually forms an abstract, realization independent system level model implemented in an SLDL like SystemC. Considering specifically data flow dominated designs, typically a process based model of computation is chosen. Following the modeling paradigm of the separation of communication and computation results in an initial system level model as e.g. illustrated in Fig. 2.9. Orthogonalization of concerns is realized in SystemC by implementing the computation in modules while communication is separated in so-called channels. For connecting modules with channels, ports and interfaces are used. The graphical notation for modules, ports and channels as in Fig. 2.9 is used throughout the rest of this work. Although bidirectional data exchange is possible, data flow dominated designs typically show an explicit data flow direction. If available, this is illustrated using specific symbols for ports and channels, see "Data flow" in Fig. 2.9.

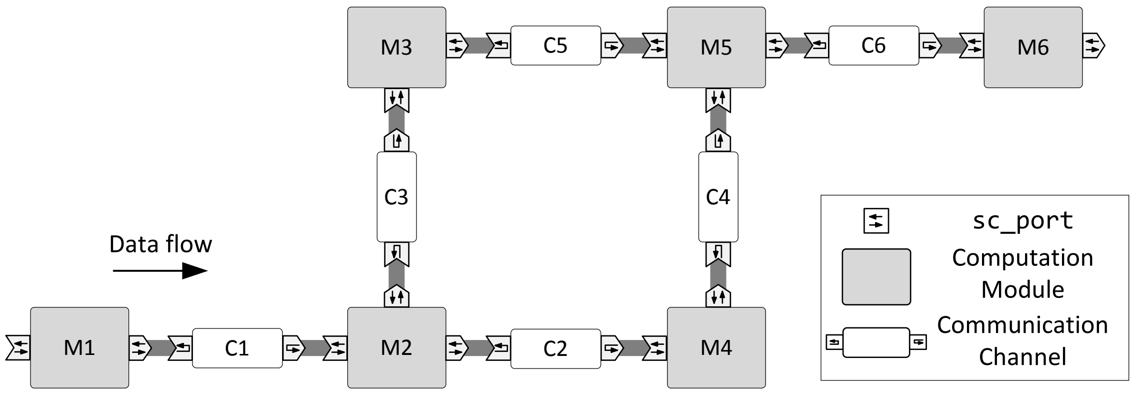


Figure 2.9: Typical high level model utilizing a process based MoC. The separation of communication and computation is realized in SystemC using modules, ports and channels.

In Sec. 1.1.4 a link to the implementation has been identified as requirement for an efficient design space exploration environment for low volume embedded systems. An important technology to overcome the gap between the system level and actual hardware implementations is the high level synthesis. It enables the translation of high level, untimed computation to RTL code. However, the analysis of a synthesizable input specification has shown that the presented high level system model is not directly synthesizable. A significantly lower level of abstraction is required especially for communication. Furthermore, HLS is strictly module based. Hence, specific module structures are required, which force the designer to give up the separation of communication and computation. Also data structures require special attention. Complex data structures which utilize dynamic memory management and heavily use pointers are not synthesizable. Furthermore, to optimally utilize memory structures in FPGAs some tools require specific code constructs.

On the implementation side, common architectures for data flow dominated, board-level systems have been analyzed. Thereby, designs including FPGAs and DSPs play a major role. The analysis once more showed the importance of a link from system level to the actual implementation and system prototypes. Both FPGA and DSP vendors provide low level tools to accurately estimate design parameters. In-system debug mechanisms like JTAG further increase the importance of early prototypes.

In summary it can be stated, that C-based system level design combined with high level synthesis forms a solid basis for design space exploration. Since also embedded software is typically based on C/C++, this leads to a complete C-based design environment. However, there are different modeling styles and requirements for the system level and for hardware synthesis and software design. To explore various application-to-architecture mappings, design components have to be shifted from hardware to software and vice versa. An efficient design space exploration is only possible if this shifting can be performed without much design effort.

3 Related Work

In this Chapter different related works concerning hardware/software codesign and design space exploration are analyzed and their advantages and disadvantages are illustrated. To support an efficient exploration of the design space especially the support of shifting design components from hardware to software and vice versa is examined.

In principle all presented works are divided into two categories: high level synthesis and electronic system level tools, see Sec. 3.1 and Sec. 3.2. The electronic system level tools focus mainly on system level design. Therefore, they typically explicitly provide capabilities for design space exploration and hardware/software codesign. However, some of these works do not provide a link to actual implementations.

The other category contains the basic high level synthesis tools, which are an essential part of a modern codesign flow, since they accelerate hardware design and they raise the abstraction level, which reduces the differences between hardware and software design. Therefore, it suggests itself to use HLS tools directly for hardware/software codesign and for design space exploration. Additionally, many HLS tool vendors promote their tools as ESL solutions [GAGS09, p. 294]. A good overview on both categories has been presented by Gajski in [GAGS09], which has been used as basis for the following outline.

3.1 High Level Synthesis Tools

In this Section different categories of high level synthesis tools are presented. Besides the predominant C and SystemC-based HLS tools, hardware accelerator synthesis tools and solutions which are based on entirely different design languages are introduced. All tools are analyzed regarding their usability for HW/SW codesign and design space exploration. Thereby, a key component is the input design language of the synthesis tool or more precisely, the subset of the design language, which is accepted by the tool. The abstraction level and the expressiveness of this language subset influences design flexibility and usability. Differences between the hardware and the software design flow complicate moving design components from hardware to software and vice versa.

3.1.1 Hardware Accelerator Synthesis Tools

If a pure software system has to be extended by a hardware accelerator due to performance reasons, a hardware accelerator synthesis tool is the appropriate solution. These tools focus mainly on FPGA systems. They support different hard- and softcore FPGA processors like MircoBlaze [64], Nios [39] and the PowerPC [26] and of course they also support the ARM platform [52]. Very often these tools provide a profiling function to identify the computation intensive parts of the algorithm. In the following three different commercial tools of this type are presented.

The first one is ESLerate from Binachip [3]. It supports solely FPGA system-on-chip solutions. Starting with a software binary, the application is analyzed and computation intensive operations and functions are suggested for outsourcing them to a coprocessor. The tool can be used with various different softcores, different ARM cores and with the PowerPC embedded on several Xilinx FPGAs. It automatically generates the required hardware interface, the software driver and it adapts the software binary accordingly.

Another tool in this category is Cascade from Critical Blue [8]. It also starts with the analysis of a software binary. Then, computation intensive parts of the algorithm are proposed for a realization as coprocessor [HT04]. Therefore, the tool provides different coprocessor architectures, which can be chosen. For simulating the whole system with the connected accelerator, a SystemC model of the new hardware block is generated. Like in Binachip's ESLerate, the coprocessor is connected via the system bus.

The third hardware accelerator synthesis tool is the Triton Builder from Poseidon [17]. It does not start with a software binary, rather with ANSI C source code, which is analyzed by a precompiler and an optimizer. Besides that, the Triton Builder provides features and possibilities comparable to the other presented tools. Although different FPGA vendors are supported, the tool's focus is clearly on Xilinx FPGAs.

The advantage of these hardware accelerator synthesis tools is that they are very automated. Only a little hardware development know-how is required to improve the performance of a software implementation by moving complex calculations to a coprocessor. Therefore, such tools are ideal for small software systems and companies with little hardware know-how. A major drawback of these solutions is the lack of flexibility. Only a few processors are supported. In the first place, special purpose processors like DSPs cannot be used with these tools. Further, the design space is limited to mixed hardware/software solutions. However, for high performance application a pure hardware realization might be the best solution. Equally, more complex systems with several chips cannot be designed using these kind of synthesis tools. As already mentioned, these tools can be used to add a coprocessor to a simple software system. They cannot be used to perform design space exploration of complex hardware/software systems.

3.1.2 C-based Synthesis Tools

An alternative to hardware accelerator synthesis tools are general high level synthesis tools. These tools have less restrictions concerning the target architecture and they can be categorized according to the accepted input language. Especially to enable the use of one source code for hardware as well as for software, the use of c-based synthesis tools seems obvious.

The first tool presented in this category is the Codeveloper [13] from Impulse Accelerated Technologies. This HLS tool could have been presented in Sec. 3.1.1 as well since it focuses especially on FPGA and hardware accelerator design. For a specified set of processors it generates hardware, software and an interface in terms of a driver and a hardware interface block. The reason for categorizing it in the general C-based synthesis tools is that it can be used as a block level

HLS tool as well. Thereby, one block is composed of a single C function. This function can be implemented in pure C. To add facilities like communication and parallelism, a proprietary application programming interface (API) is provided. Communication is realized using predefined interface blocks, which are mainly based on streaming and shared memory. Further, synthesis constraints can be defined using `#pragma` directives.

Another C-based HLS tool is Synphony from Synopsys [21]. Originally developed by Synfora under the name Pico, it was acquired by Synopsys in 2010. A characteristic of Synphony is the configurable architecture template [GAGS09, p. 324], [CM08, p. 53]. Each design is mapped to this architecture template. It consists of so-called processing arrays. The data exchange between those processing arrays is controlled by the timing controller and performed via raw signals, shared memory or FIFOs. To connect a designed block to other components, the tool provides a small set of interfaces. The limitation to a manageable set of interfaces may simplify interface verification for the designer. However, these interfaces and the more or less fixed architecture restrict the design flexibility.

The third HLS tool presented here is eXCite from YXI [25]. This tool also restricts the input language to ANSI C/C++. It performs block level synthesis, whereby one hardware block is generated from one C/C++ function. Comparable to the Codeveloper, the tool provides a proprietary API with a set of predefined communication interfaces. It is not possible to design user defined communication components. Like most synthesis tools, it does not support software features like dynamic memory management and pointer arithmetic.

Obviously, the possibility to directly generate hardware from C/C++ code is interesting. It enables one source code, which can be used for hardware and for software design. Whereby, of course a specific coding style is required, since not the whole C/C++ standard is supported. An example for unsupported constructs are complex data structures, dynamic memory management and pointer arithmetic, which have to be replaced by synthesizable constructs. These restrictions limit the possibility to have one source code for hardware as well as for software.

Another disadvantage specifically of C-based synthesis tools is the lack of flexibility. The designer has to get along with a set of predefined computation components. The realization of special user defined communication interfaces is only possible via communication adapters in VHDL or Verilog or via proprietary language extensions. Such language extensions in turn lead to a tool dependent source code. The limited flexibility also concerns the modeling of parallelism. Pure C/C++ does not provide any facilities to model parallel computations. This significantly reduces the modeling capabilities. Further, it increases the importance of the ability of synthesis tools to recognize and exploit the inherent parallelism of algorithms.

The direct synthesis of an algorithm in C/C++ denotes a fast and simple way from an abstract software code to a hardware implementation. However, the design of heterogeneous hardware/-software systems often requires models at different abstraction levels. Often it is necessary to simulate several design components at different levels of abstraction simultaneously, to verify the functionality of the whole system. C/C++ does not support such a range of different abstraction levels and it also does not provide facilities for such system simulations. This kind of modeling and simulation is only possible via system level design languages like SystemC or SpecC.

3.1.3 SystemC based Synthesis Tools

High level synthesis tools, which accept a system level design language as input, mostly rely on SystemC. In the following six different SytemC based HLS tools are briefly presented. One of them, the Cynthesizer from ForteDS [49] is a pure SystemC synthesis tool, while the other tools support inputs in pure C/C++ as well.

The Cynthesizer from ForteDS is one of the oldest HLS tools [GAGS09, p. 323], [CM08, p. 75]. It supports the synthesis of untimed algorithms. Whereby, the input and output behavior can be modeled using a cycle- and pin-accurate protocol. ForteDS also provides pre-defined interfaces like hand shaking interfaces in terms of proprietary ports. However, the tool also has several restrictions. Like most HLS tools, it does not support dynamic memory management and pointer arithmetic. Simple arrays can be mapped to memories or registers. If several data structures have to be mapped to one and the same memory structure, an explicit memory model, which is generated via the Cynthesizer, has to be instantiated.

Another relatively new tool is C-2-Silicon from Cadence [33]. Its features are comparable to the ForteDS's Cynthesizer. It also has restrictions concerning data structures, e.g. dynamic memory management and pointer arithmetic are not supported. However, it at least partly supports merging arrays to better exploit available memory resources without requiring the explicit instantiation of memory models.

The third presented tool has recently been acquired by the FPGA vendor Xilinx [23]. It was originally developed at UCLA with the name XPilot [GAGS09, p. 346]. Now it is offered by Xilinx as high level design tool called AutoESL. Like the other SystemC tools mentioned, it supports user-defined, cycle-accurate interface descriptions for SystemC designs. If pure C/C++ code is synthesized, pre-defined interface options can be used [CM08, p. 99]. It also provides the possibility to merge different arrays to one memory block. However, the tool does not allow the definition of user defined data structures using classes and like most HLS tools it does not support dynamic memory management or pointer arithmetic.

Other SystemC based HLS tools worth mentioning are Catapult C [5] from Calypto and Cyberworkbench from NEC [15]. Both have similar capabilities compared to the other tools, which have already been presented. Whereby, the Cyberworkbench is not only a synthesis tool, rather it is a tool suite, which also provides facilities for cosimulation and verification.

Several advantages of the SystemC synthesis tools result from advantages of SystemC itself. An example thereof is the possibility to combine design components at different abstraction levels in one system simulation, which is a major benefit of SystemC. Most of these tools support predefined communication components. Using cycle- and pin-accurate interface descriptions, it is however possible to realized user-defined interfaces. This keeps the design simple but provides flexibility if needed.

Obviously SystemC HLS tools focus mainly on hardware design. This leads to disadvantages if these tools should be integrated into an ESL environment. Hardware specific code structures as used e.g. for modeling an interface cycle and pin-accurate lead to platform dependent source code. This complicates reusing the source code for software design. Another example is the mapping of data structures to memories, whereby ForteDS requires the instantiation of an explicit memory component. Further, the restrictions concerning dynamic memory management and pointer arithmetic limit the reusability of algorithms as well.

3.1.4 Other Solutions

Although the majority of HLS tools support C/C++/SystemC, different other approaches exist as well. In the following, three different high level synthesis tools of this category are presented.

Like in the other categories, their applicability for hardware/software codesign and design space exploration is especially analyzed.

The first tool presented is the Bluespec Compiler [4]. It is a serious competitor to the C/C++/-SystemC high level synthesis tools if we consider pure hardware design [GAGS09, p. 326], [CM08, p. 129]. The compiler accepts input descriptions in Bluespec System Verilog, which is based on the System Verilog [oEI05] syntax. The core technology is based on a term rewriting system, whereby terms describe hardware states and rules describe behavior. The compiler generates Verilog RTL for synthesis and SystemC for simulation. The language and the compiler especially target hardware with complex control logic like memory controllers and I/O peripherals. Its focus and the used input language show that Bluespec is intended for hardware design. The system is modeled at a significantly higher abstraction level compared to RTL designs. However, since the modeling concept is very different compared to software design, a utilization of Bluespec for codesign is difficult.

In contrast, the Codetronix Mobius [6] especially addresses the generation of hardware and software from one and the same source code. To realize communication between hardware and software components, the Mobius compiler also generates bridges for Xilinx, Altera and ARM processors. The input language, also called Mobius, is based on Pascal with multi-threaded extensions using the communicating sequential processes methodology [uAS07]. The compiler translates the input description into VHDL, Verilog or ANSI C. For Xilinx FPGAs, the tool XPSUpdate even generates a design project for the Xilinx development environment including all generated design components. However, the main drawback is probably the input language, since high level language features like templates or complex data structures are not supported. These features significantly simplify system design and design reuse.

The third example in this Section is the FalconML synthesis tool from Axilica [2]. It is an example for a tool enabling the synthesis of UML [55] designs. The structure is modeled using the SysML UML profile and C or C++ can be chosen as action language. The models can be translated to C or C++ code or to an RTL hardware description. For high performance functional simulation a SystemC model can be generated as well. Indeed, the approach to start with UML as system level design language is interesting, since UML is the de facto standard for the specification of software designs. However, UML lacks many capabilities a more hardware related system level design language such as SystemC possesses. This software oriented approach does not allow the modeling of the system on different abstraction levels, neither does it allow the combination of different abstractly modeled design components in one simulation. Therefore, it becomes difficult to perform optimizations and refinement for specific target architectures.

These three examples show that there are indeed interesting alternatives to the C-based HLS solutions. However, as mentioned in [GAGS09, p. 327], the most popular tools are based on C and SystemC. A reason this is certainly the wide distribution of C/C++ as software programming language. This simplifies the usage of C and SystemC for many designers. Especially for designing embedded systems, C/C++ is the predominating programming language. Therefore, the use of a completely different design language for system level design only introduces additional translation effort and it reduces the interoperability of design components in different design phases.

3.2 Electronic System Level Tools

Sec. 3.1 has shown that high level synthesis tools are an essential part of a codesign flow, however they do not provide sufficient capabilities to perform efficient design space exploration (DSE) and codesign. In this Section so-called electronic system level tools are analyzed in more detail. These tools focus especially on codesign and DSE and some of them integrate HLS tools for translating the high level code to a hardware implementation. Hence, they are supposed to solve the difficulties, which arise if solely HLS tools are used for codesign. A major part of these tools exist solely as academic solutions, which are presented in Sec. 3.2.1. Commercially offered ESl tools are analyzed in Sec. 3.2.2.

3.2.1 Academic Solutions

The academic electronic system level solutions typically consist of a set of tools, which provide facilities to perform tasks like architecture exploration, hardware and software synthesis and prototype generation. In this Section four different examples of such ESL solutions are presented and analyzed regarding their applicability for hardware/software codesign and design space exploration. All of these tools have their strengths in different areas. However, they all have limitations and restrictions, which reduce the applicability of these tools.

3.2.1.1 SystemCoDesigner

SystemCoDesigner is a tool for fully automated design space exploration and rapid prototyping of multiprocessor SoCs (MPSoC). The tool uses SystemC and a library called SysteMoC [FHT06] for modeling the system. The high level synthesis tool Cynthesizer [49] from ForteDS is integrated into the design flow and enables the generation of custom hardware blocks. Design metric estimation is performed using synthesis tools and simulation of compiled source code. Promising architectures can be automatically implemented on an FPGA for rapid prototyping.

The design flow of the SystemCoDesigner shown in Fig. 3.1 starts with a behavioral model using SysteMoC. This is a SystemC library, which supports the usage of a model of computation called functions driven by state machines or funstate [STG$^+$01]. Each module has only one thread. In this thread a state machine is implemented, which defines the communication behavior of the module. The modules, also called actors, are connected using SystemC FIFOs. The computation parts in each module are implemented in methods. In this way computation is separated from communication.

Each module can be transformed to a hardware accelerator. Therefore, the computation methods are automatically transformed to a synthesizable SystemC module and the HLS tool Cynthesizer is used to translate this module to an RTL description. By using different design constraints different hardware accelerators can be generated for the same actor. Finally, a VHDL file is generated, which instantiates the controlling FSM and the synthesis output from the HLS tool. This file is then further synthesized using a logic synthesis tool. The resource and performance information provided by the logic synthesis tool is required by the automatic design space exploration. The actors can also be transformed to software by simple code transformation. The different hardware accelerators and the software implementation together with performance and resource information are then added to the component library.

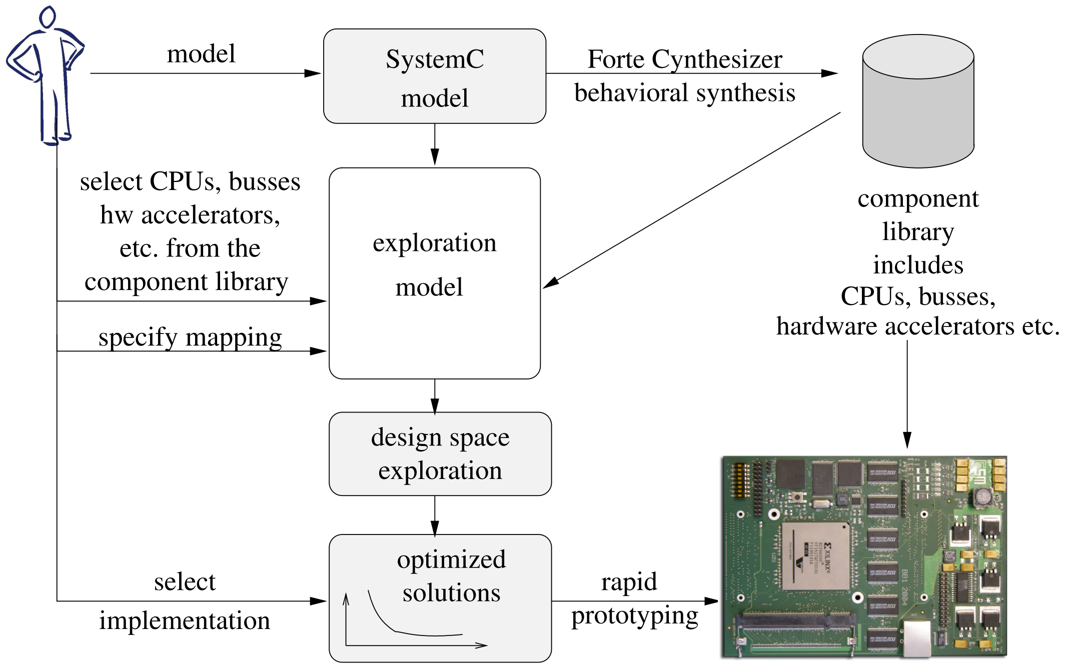

model
SystemC model
Forte Cynthesizer behavioral synthesis
select CPUs, busses hw accelerators, etc. from the component library
exploration model
component library includes CPUs, busses, hardware accelerators etc.
specify mapping
design space exploration
select implementation
optimized solutions
rapid prototyping

Figure 3.1: Design flow using SystemCoDesigner [HSKM08]. Each design component can be translated to a hardware accelerator using high level synthesis. These hardware accelerators and other elements from the component library are used to generate different design solutions, which are automatically compared during design space exploration.

Beside the hardware accelerators, the component library also includes synthesizable intellectual property (IP) cores like CPUs, buses and memories. These components together with a so called architecture template are used to explore the design space of the given application. This exploration is performed automatically using multi-objective optimization algorithms to find optimized HW/SW solutions. Design metrics like performance and resource information are provided by synthesis tools, obtained by simulation or taken from the IP core data sheet. Finally, the best solutions can be automatically realized on an FPGA using Xilinx soft core processors for rapid prototyping.

The SystemCoDesigner has some limitations. It is restricted to multi media and networking, i.e. streaming applications. Communication can only be modeled using SystemC FIFOs. This is not a problem as long as only data flow dominated systems are realized using the SystemCoDesigner, but modern systems which might include data and control flow dominated parts cannot be realized with this codesign environment. Also the Algorithm modeling is restricted to the language constructs, which are synthesizable by the Forte Cynthesizer. Hence, complex data structures cannot be used, which significantly decreases the abstraction level.

3.2.1.2 System-On-Chip Environment

The System-On-Chip Environment is a system level design tool developed at the University of California, Irvine [DGP+08]. It supports embedded systems design starting with a specification modeled in SpecC[GD01]. Interactively guided by the user via a graphical user interface, the specification is refined step by step to an actual HW/SW implementation. The tool allows the exploration of different design solutions at various levels of abstraction and allows the automatic model refinement to lower abstraction levels. The main focus is the design of heterogeneous multi processor system on chip solutions.

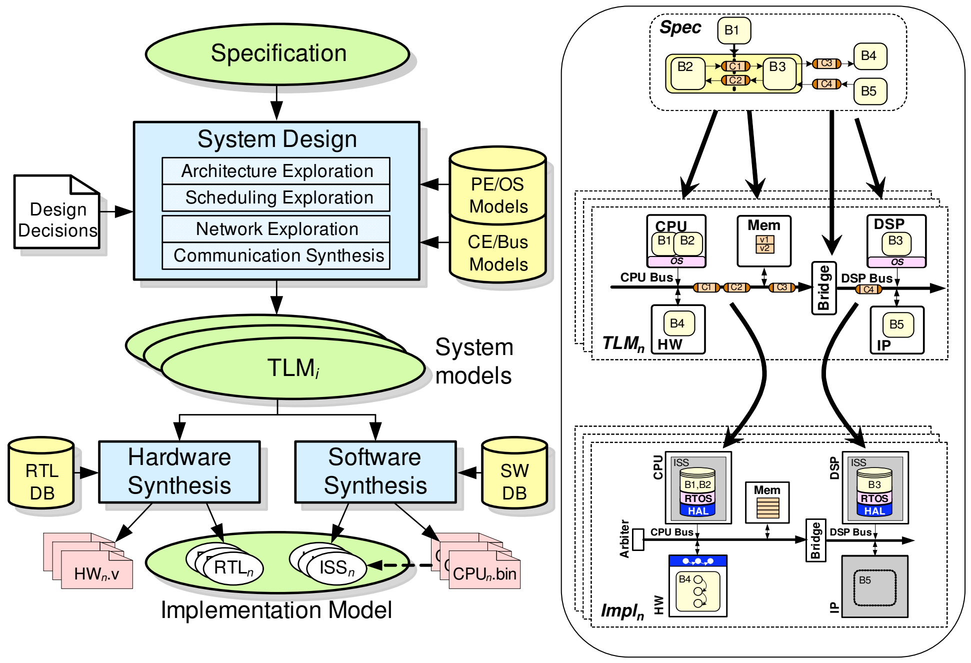


Figure 3.2: System-On-Chip Environment design flow [GHP+09]. The front end translates the specification into different transaction level models (TLM), which synthesized to actual implementations by the back end.

The basic design flow of the System-On-Chip Environment is shown in Fig. 3.3. It basically consists of a front end and a back end part. The front end generates different transaction level models(TLM) out of a specification. The synthesis of hardware and software implementations out of a given TLM is handled by the back end.

The input specification is modeled using SpecC. Similar to SystemC, SpecC allows hierarchical and concurrent modeling of different computation components. In SpecC these components are called behaviorals. Each behavioral represents a behavioral description of a basic algorithm in ANSI C. The different computation components are connected via channels, which represent the communication among the behaviors. A standard library provides frequently used channels like synchronous or asynchronous message passing, events and queues.

The system specification is then successively refined to a model at the transaction level. The refinement process follows the Specify-Explore-Refine methodology [GVNG94]. In each refinement step, different solutions to a design decision are explored. Retargetable profiling and estimation tools provide feedback to find the decision, which fits best to the given design constraints. A plug in mechanism is also available to add algorithms for automated decision making.

The first step of this refinement phase is the architecture exploration, where the different behaviorals are mapped to different processors or to custom hardware blocks, which are provided by a library. The computation components on a processor run above an abstract operating system. In the scheduling exploration phase, different scheduling algorithms are compared and evaluated. During network exploration and communication synthesis, the given communication channels are mapped to communication elements like bus models and bridges or transducers.

The back end part of the design flow handles the actual hardware and software synthesis. Behavioral computation components mapped to a custom hardware block are translated to RTL using high level synthesis. The software synthesis automatically generates embedded software code for each processor of the system model.

The System-On-Chip Environment is a very complete automated design framework. However, a major drawback is the input language SpecC. In recent years SystemC emerged as the industry de facto standard for modeling and simulation. This might significantly reduce the industry acceptance of this design environment. The use of SpecC, which is based on ANSI C also reduces the design and modeling capabilities. Algorithms, which are modeled using a high level language may operate on complex data structures. To use the design flow in this case, a manual translation to SpecC using only simple data structures is required.

3.2.1.3 Daedalus

The Daedalus framework presented in [NSD08] is a design framework for system-level architecture exploration, system-level synthesis and prototype generation using a set of tools. It guides the designer from a sequential C program to an heterogeneous multi processor system-on-chip solution.

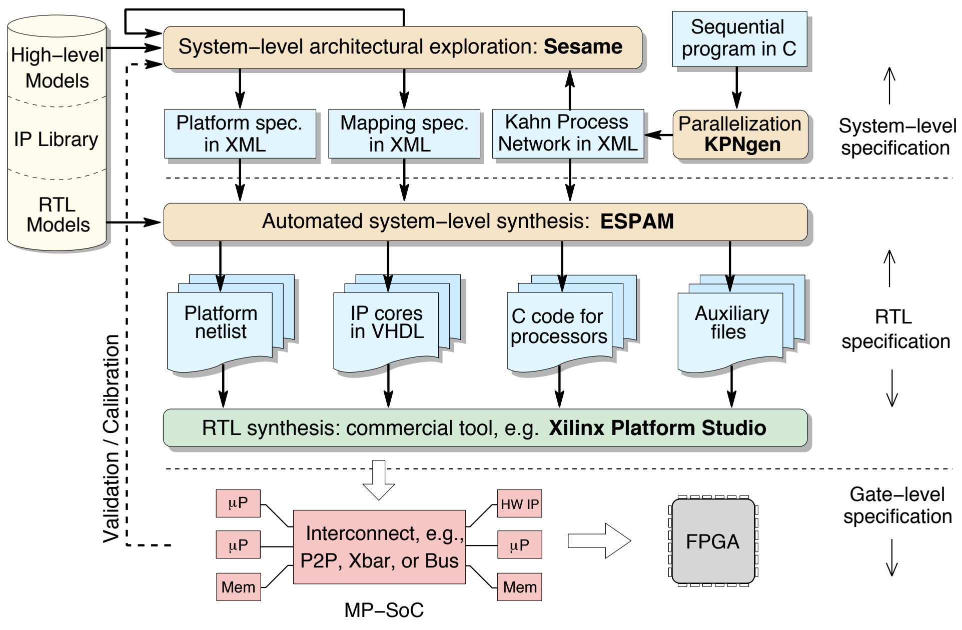


Figure 3.3: The Daedalus design flow [GHP+09]. Starting with a sequential C code, architecture exploration, system-level synthesis and prototype generation is performed using the tools KPNgen, Sesame and ESPAM.

The design flow of the Daedalus framework is shown in Fig. 3.3. The input specification has to be implemented as a sequential C program, which is then translated to a concurrent Kahn process network (KPN) [Kah74] using KPNgen. It focuses only on data flow dominated applications and is restricted to programs which are specified as static affine nested loop programs. KPNgen analyses the input specification and performs a parallelization. Behavioral design space exploration can be performed by varying the amount of parallelization.

After that, an automatic design space exploration using the Sesame tool is performed. Sesame is a modeling and simulation environment, which allows architecture exploration by mapping the KPN to MPSoCs composed of IP library components such as processors and dedicated hardware components. The models can be gradually refined to increase simulation accuracy. Additionally, Sesame supports heuristic search methods and a design space pruning step to trim the design space. The output of this architecture exploration phase is a set of promising architecture candidates, which are described using a XML-based platform description.

These high level descriptions and the KPN generated by KPNgen are the input to the third tool called ESPAM, which replaces the high level models with RTL models from the library and generates C code for the processors. The RTL models of the processors and dedicated hardware blocks together with a generated platform net list can directly be synthesized using logic synthesis tools and can be mapped to FPGAs for prototyping.

A disadvantage of the Daedalus framework is the restriction to C. Although the framework provides a fast way from an algorithm in pure C to a HW/SW prototype, its modeling capabilities for heterogeneous systems are very limited. Heterogeneous systems which consist of a data flow dominated and a control flow dominated part cannot be modeled with the Daedalus framework.

Another disadvantage of C is the lack of high level features like complex data structures, which significantly increase the abstraction level and are frequently used by algorithm designers. The whole framework focuses mainly on comparing different multi-processor architectures and on the mapping of software parts to different processors. Modern HW/SW codesign requires greater flexibility in terms of dedicated hardware generation. This flexibility would be provided by a high level synthesis tool, which enables the generation of specific hardware blocks to speed up the implementation of a particular algorithm. In the Deadalus framework, the generation of dedicated hardware is limited to library components.

3.2.1.4 CATtree

Another refinement based design environment is presented in [PYC06a]. This SystemC based design framework enables TLM-RTL-SW cosimulation including different levels of abstraction. It also simplifies the generation of different HW/SW prototypes.

The design flow from the system level to a prototyping implementation is shown in Fig. 3.4. The approach strictly follows the separation of concerns philosophy and separates communication from computation at all abstraction levels. For communication refinement towards a hardware and a software implementation, the CATtree library [PYC06b], a library with communication components at different abstraction levels has been developed. At the system level communication primitives like FIFO, Array, Event and Bus are available, which represent only the functional behavior of a communication link. During refinement these primitives can be replaced by components, which add more details about the communication architecture. At each refinement step different components on a lower abstraction level are available. This tree-like structure of the library explains the library's name. Finally, fully implemented communication components from the CATtree library are used. These components are configurable RTL or software implementations of the high level communication primitives.

Computation components also have to be replaced by refined components. For hardware, these components are replaced by RTL implementations, which are either available from a previous project or have to be coded by hand. A high level synthesis tool might be used for an automatic translation, but none of these tools is actually integrated into the design flow. Mapping computation components to software is handled using the DEOS operating system, which provides SystemC APIs. Thus, it is possible to directly compile SystemC Modules for the desired platform.

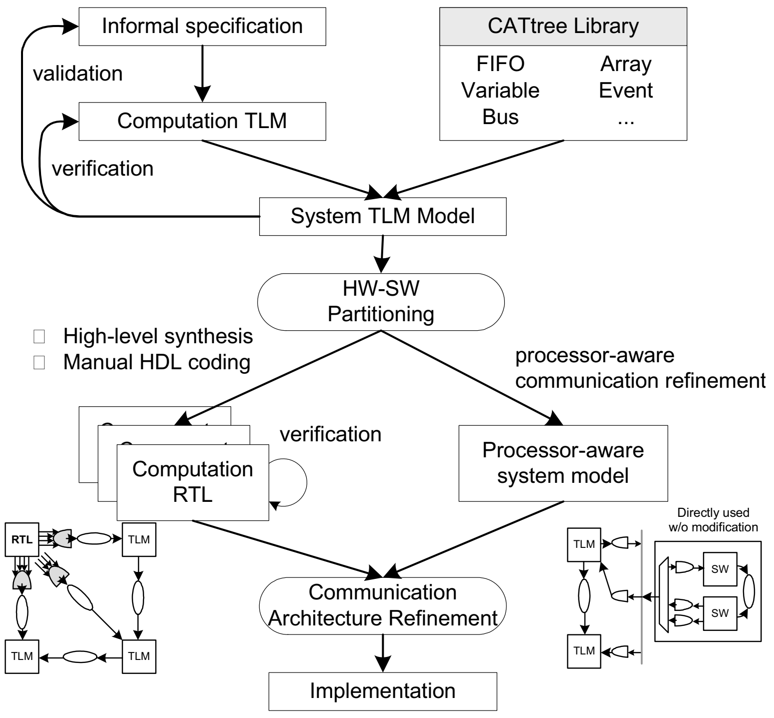


Figure 3.4: The CATtree design flow [PYC06a]. A library based approach is used to successively replace high level communication components by refined elements. Computation components have to be refined manually or by using a currently unintegrated HLS tool.

A disadvantage of this approach is the missing integration of a high level synthesis tool. A manual translation from a system level model to an RTL implementation significantly reduces the applicability of this approach. It introduces a large design effort if different HW/SW solutions are generated and compared during design space exploration. Although they claim the possibility to translate a behavioral computation component to an RTL implementation using a C-to-RTL tool, the integration of such a tool would introduce a lot of restrictions. Furthermore, the tool has to generate an IP core with a pin interface compatible to the interface of the RTL components of the CATtree library. Additionally, many tools require specific code structures at the system level. It seems to be a real challenge to find a tool, which accepts the used input format and produces the required RTL implementations with the CATtree interfaces.

3.2.2 Commercial Solutions

Most of the currently available commercial ESL solutions especially target the early architecture exploration by providing facilities for easy architecture modeling and application-to-architecture mapping via drag-and-drop. Four examples for commercial ESL solutions are presented below. Again, their applicability for hardware/software codesign and design space exploration is analyzed.

An example of a commercial ESL tool is the CoFluent Studio, which was acquired by Intel in 2011 [7]. It provides a modeling and simulation environment with a graphical front end for capturing the system behavior using SystemC [GAGS09, p. 329]. The application is modeled as a network of timed processes. The environment provides an initial performance and functional evaluation. After defining the architecture graphically, the application elements can be mapped to the specified platform via drag-and-drop. The CoFluent Studio then automatically generates a SystemC TLM simulation model. The tool allows a fast simulation of different specification to architecture mappings at early design stages. However, it does not support model refinement towards implementation or HW/SW prototypes.

Another commercial ESL solution is a tool suit named Platform Architect from Synopsis [16]. Originally developed as internal project at the Interuniversity Microelectronics Centre(IMEC) in Belgium [GAGS09, p. 330]. The research results have been incorporated into the products from a company called CoWare. Finally in 2010, Synopsis acquired CoWare and its products. The Platform Architect tools focus on architecture exploration, platform design and embedded software development. They provide a graphical environment for system capturing and a modeling library with a lot of hardware IPs, programmable processors and system buses. Like the CoFluent Studio, these tools focus merely on simulation and modeling. They do not support the designer with a link to the actual HW/SW implementation.

The third presented commercial electronic system level tool is the SoC Designer from Carbon Design Systems [19]. It enables platform architecture capture and modeling via graphic user interface [GAGS09, p. 331]. The environment allows cosimulation of SystemC, VHDL, Verilog and Matlab sources. A fast cycle accurate simulation is achieved by statically scheduling all simulation components before execution. The core simulation technology behind this tool has been developed at the University of Aachen in Germany. This simulation methodology simplifies virtual prototype generation and it enables architecture analysis and exploring the performance of different hardware/software trade-offs. However it does not support the design of custom hardware components. Comparable to the other commercial ESL tools presented, it does not provide a link to actual implementations. Only low level implementations and actual prototypes can provide accurate design parameter estimates for different design solutions.

The last ESL tool called SpaceStudio from Space Codesign [20] is the only tool presented here, which provides a link to actual hardware implementations by the possibility to integrate an HLS tool like the Cynthesizer or Catapult C. The name Space is an acronym for SystemC partitioning of architectures for co-design of embedded systems and the company Space Codesign Systems has been founded as a spin-off from the Ecole Polytechnique de Montreal [FCM+07]. The main difference to the other tools presented is the possibility to integrate a synthesis tool. Besides that, it is a system level integration development environment based on Eclipse [GAGS09, p. 329]. The system architecture can be set up graphically and the application-to-architecture mapping can be performed using drag-and-drop. System performance can be evaluated via automatically generated SystemC TLM models. The so-called GenX technology provides automatic FPGA prototype generation by replacing hardware IPs with pre-designed RTL implementations. Custom hardware blocks are synthesized using the integrated HLS tool.

The disadvantage of most of the presented commercial ESL tools is that they do not provide a link to the implementation. Only such a link guarantees a simple generation of prototypes and an accurate resource and performance estimation using these prototypes. Only SpaceStudio provides such a link by the possibility to integrate an HLS tool for hardware generation. However, this integration leads to limitations caused by the synthesis tool as already presented in 3.1.3. The efficient use of data structures and I/O protocol implementations require domain specific code, which cannot be shifted from hardware to software or vice versa without manually adapting the source code.

3.3 Summary and Evaluation

In the previous Sections different related works have been presented and analyzed regarding their applicability for design space exploration of board-level systems. As already mentioned a key requirement is the easy shifting of design components from hardware to software and vice versa. Obviously, the smallest design effort is required if the tool fully automates hardware and software generation. Examples therefore are the hardware accelerator tools. However, this degree of automation is only possible if a very limited set of architectures is supported. Hardware accelerator tools solely focus on processor plus coprocessor designs.

More flexibility regarding the hardware architecture is provided by general HLS tools with C as input language. They are applicable for high level algorithm synthesis. However, C is a pure software programming language and does not provide facilities for modeling communication interfaces. Therefore, tools provide non-standard language extensions or they are limited to a set of predefined interfaces. Both solutions have their disadvantages because they lead to tool dependence and reduce flexibility.

The most flexibility for hardware design is provided by SystemC based HLS tools. They can be utilized to generate various hardware architectures with user defined communication interfaces. However, a lot of refinement effort is required to get from a high level model to a synthesizable SystemC description. Thereby, especially data structures have to be mentioned. Complex data structures utilizing dynamic memory management and pointer arithmetic are not supported at all by most tools. They have to be replaced by low level data structures. To efficiently utilize FPGA memory structures it is sometimes even necessary to use specific code constructs. This illustrates that HLS tools focus mainly on hardware design and do not provide facilities to perform actual hardware/software design space exploration. Therefore, specific ESL solutions are required.

These ESL solutions have been presented in two categories: commercial and academic tools. The first category supports the designer during architecture exploration via modeling and simulation facilities. Most of them do not integrate HLS tools into their environment. Hence, they focus purely on the system level and do not provide a link to actual implementations. In contrast, most ESL solutions of the second category integrate an HLS tool for hardware generation. As a consequence, many restrictions of the HLS tools also apply to these ESL solutions. Thus, they also have limitations regarding complex data structures, which significantly limits the supported abstraction level. Another disadvantage of some existing solutions is the limitation to a certain application domain which reduces their applicability.

Following the analysis of related work, the best solutions are the academic ESL tools. They provide a very complete design environment with a link to actual implementations. However, there is no solution which directly supports a high level functional model including complex data structures. Especially more complex algorithms utilize high level data structures. To map such algorithms to hardware or software a designer has to replace them with hardware or software specific data structures. This denotes a significant effort if during design space exploration various application-to-architecture mappings are evaluated. A sufficient ESL solution has to support complex data structures and it has to support their efficient mapping to the target architecture.

4 New Tripartite System Design Approach

The analysis of existing electronic system level design environments has shown that none of them fulfills all demanded requirements. The limitation to simple data structures has been identified as crucial restriction which limits the level of abstraction. The goal is to develop an ESL environment which significantly raises the abstraction level by directly supporting complex data structures.

In the following, a hardware/software codesign flow called the Tripartite Design Approach is presented. It enables the realization independent design of computation modules despite the usage of complex data structures. This is realized by separating not only communication, but also data structures from computation. The so-called HWSW-Module, which implements only computation, can be mapped to efficient hardware and software implementations without changing the code itself. The realization independent design of a computation module is presented in Sec. 4.1 in detail.

For the mapping to hardware or software, communication components and complex data structures are replaced by refined implementations. Thereby a library based approach is chosen, which enables the reuse of often used design components. At the same time, a library based approach simplifies the extension with application specific design components. This enables the utilization of this approach for various different applications. A library providing often used complex data structures, the Codesign Template Library (CTL), is presented in Sec. 4.2.

Eventually, in Sec. 4.3, the system design using the Tripartite Design Approach (TDA) and the Codesign Template Library is illustrated. Thereby, the system modeling, the generation of different HW/SW partitionings and the evaluation of the generated design solutions are presented in more detail.

4.1 A Realization Independent Computation Module

Design space exploration of hardware/software systems can be significantly simplified if the different functional units can be designed realization independent. Hence, if the functional unit can be reduced to its fundamental functionality without making assumptions about the actual implementation target. Concurrently a methodology is required, which transforms this realization independent functional unit to implementations, which exploit the characteristics of the respective target architecture.

For developing such a realization independent module the appropriate system level language has to be chosen. In Sec. 4.1.1 the advantages and disadvantages of the different design languages are compared. It becomes apparent that SystemC offers many modeling capabilities for hardware design and is still very close to software since it is basically a C++ library. However, due to modeling differences, it is not possible to use one and the same SystemC module for high level simulation, hardware and software design. These differences are illustrated in Sec. 4.1.2. To overcome these modeling differences, a module style and a preprocessor macro system is presented in Sec. 4.1.3. The result is the HWSW-Module, a computation module, which can be adapted for the high level simulation, hardware and software design.

Especially complex data structures lead to realization dependent code. Efficient synthesizable hardware code often has to explicitly use memory structures available on the target platform. On the other side, dynamic memory management and heavy use of pointers, which is typically used to implement complex data structures in software, is not accepted by most HLS tools. One way to solve this problem is the introduction of the software concept of abstract data types for system level design, which is presented in Sec. 4.1.4. Exploiting this concept, data structure implementations can be separated from their interface. High level, hardware and software specific implementations can be exchanged easily. A basic overview of the resulting Tripartite Design Approach is presented in Sec. 4.1.5. Finally in Sec. 4.1.6, restrictions and limitations of the presented design flow are investigated.

4.1.1 An Appropriate System Level Design Language

An important point of a HW/SW system level design approach is the chosen design language. In the past different languages have been proposed for system level design, like the object oriented approach in Ptolemy [KL93] or the extension of VHDL in [SP94]. Recently, mainly C-based languages are used. The system level design language influences both the design possibilities and the design complexity of a design flow. In the following, the appropriate system level design language for a realization independent computation module is chosen.

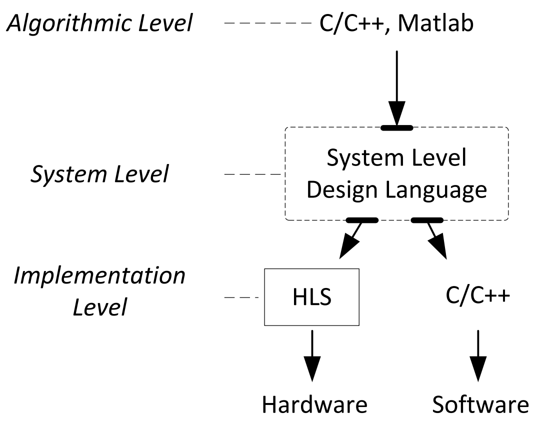


Figure 4.1: The system level has three different interfaces to other design domains, one to the algorithmic level and two to the implementation level.

The SLDL is used to model systems on different abstraction levels. Algorithms are typically developed using C/C++ or Matlab [14]. At the system level those algorithms are used to build the functional specification of the system, which is then successively refined. For the actual hardware or software implementation different design languages are used. Hardware is typically modeled using VHDL or Verilog. High level synthesis provides the possibility to translate system level languages into VHDL or Verilog. For the implementation of embedded software very often C is used. DSP software for example is solely implemented in C, only very time critical parts are realized using assembler [Keh05]. Modern DSPs also come with C++ support, e.g. [63].

The system level basically has three interfaces to other design domains, see Fig. 4.1, one interface to the algorithmic level, one to hardware and one to software design [BHS09]. An interface to another design domain might require the translation to another design language. Since every manual translation between design languages is an error prone process, a system level design language should be chosen, which minimizes the required translations. Following the fact that many algorithms are available in C/C++ and that compilers exist for many software platforms, it is natural to choose a C-based system level language.

As presented in Sec. 3.1 C-based languages are the most common input languages for HLS tools as well. Tab. 4.1 shows a selection of HLS tools. All big players of the EDA industry (see Synopsys, Mentor Graphics, Cadence in Tab. 4.1) offer tools which translate either C/C++ or SystemC to a hardware description. Except for the solution from Synopsis all listed tools support SystemC as input language. The Xilinx tool AutoESL, currently the only HLS tool provided by an FPGA manufacturer, also supports C/C++ and SystemC.

Table 4.1: High level synthesis tools of major EDA companies. The input column lists the design languages supported by the respective tool.

Synthesis Tool	Company	Input
AutoESL	Xilinx	C/C++/SystemC
Synphony C	Synopsys	C/C++
Catapult C	Mentor Graphics	C++/SystemC
Cynthesizer	ForteDS	SystemC
C-to-Silicon Compiler	Cadence	C/C++/SystemC

This list indicates that C-based SLDLs are very commonly used. Since most tools support C/C++ and since many algorithms and embedded software are developed in C/C++, it seems obvious to use C/C++ as SLDL. However, the problem in this case is the lack of possibilities to model the structure of the system. C/C++ has no facilities to model hierarchy, timing or interfaces between different tasks. Some of the tools try to compensate these disadvantages by adding proprietary language constructs. Since these language constructs are non-standard, the code becomes tool specific and the advantage of C/C++ as SLDL is lost. Another approach is to automate the interface generation with the HLS tool itself. In this case, the flexibility of the designer is decreased significantly. A major advantage of SystemC is the possibility to model systems at different abstraction levels. Thereby, systems can be refined successively without changing the design language. This advantage is lost if pure C/C++ is used at the system level.

To keep some flexibility for optimizing the structure of the system, a C-based SLDL with hardware description language facilities is preferable. By choosing a standardized C-based SLDL the interoperability with other tools is assured and vendor dependencies are avoided. The most popular C-based SLDLs are SpecC and SystemC. They both have similar features and capabilities. The main drawback of SpecC is the lack of tool support. There are no industry ready HLS tools on the market. Therefore, SystemC is chosen as SLDL in this work.

4.1.2 Modeling Differences

Following the advantages described in Sec. 4.1.1, it has been decided to use SystemC as system level design language. From the three different interfaces of the system level, three different models emerge: the high level model, the synthesizable hardware model and compilable software model, see Fig. 4.2. Although SystemC is based on C++, the usage of SystemC leads to modeling differences in these three models. The high level model is the true realization independent model, which abstractly models the functionality of the given design component. To use the tools for hardware and software generation, refined hardware and software specific models are required. On the hardware side, the HLS tool requires some specific format, which has been presented in Sec. 2.2.2. On the other side, for software generation, the compiler needs a pure C++ model. To easily move components from hardware to software it is required to be able to translate the high level model to a refined model with minimal effort. Ideally, this is possible without changing the source code of the algorithm. In the following, the modeling differences between these three different models are analyzed in more detail.

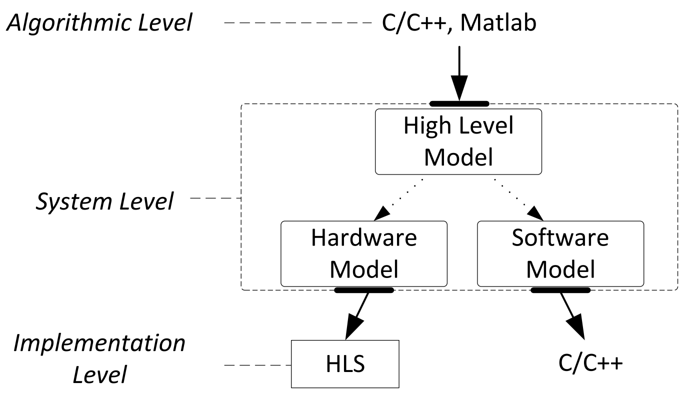


Figure 4.2: Three different models of one and the same design component at the system level. The different models emerge from the different modeling requirements by the three interfaces of the system level.

A typical SystemC high level model has already been presented in Sec. 2.1. Due to the separation of communication and computation, a SystemC module consists solely of computation, which is implemented in one or more parallel processes. Usually, these processes are realized using the SystemC process `SC_THREAD`. Basically, other process types like `SC_METHOD` can be used as well. However, the `SC_THREAD` is intended for high level modeling. The communication is separated in channels, which are connected via an `sc_port`. Thus, communication in the modules is solely realized using function calls to the port. These functions are actually implemented in the channels. In the high level model the communication is typically realized using abstract communication mechanisms provided by SystemC, e.g. events. The computation in the modules is typically implemented completely untimed. To simulate a realistic execution time, wait functions, which delay the simulation by an explicit time, can be included. Since the algorithm is often developed using C/C++ and since most SystemC data types are specifically for modeling hardware, simple C/C++ data types are used in the high level model.

Such a high level model is not accepted by a synthesis tool. In Sec. 2.2.2 a typical synthesizable SystemC module has been derived using the user guides of several different HLS tools [Fin10], [44, 42, 50]. For all these tools an `SC_CTHREAD` has to be used to implement computation processes. Further, each module has to have a clock and a reset input. Typically, the data types are refined for the hardware implementation. Although the standard `int` data type is synthesizable, it is usually replaced by data types with arbitrary bit width to reduce the hardware effort to a minimum. Like in the high level model, the computation can be implemented untimed. Controlled by design constraints, the high level synthesis tool generates an FSMD, which implements the specified computation. A significant difference between a high level and a synthesizable module is that the separation of communication and computation has to be given up. This is explained by the interface requirements for a synthesizable module. In contrast to the abstract, untimed function interface in a high level module, a synthesizable module has to have a bit and cycle accurate interface. Hence, the input and output ports represent the same bit interface as the later generated RTL model. Each communication and synchronization of a module with other design components requires a cycle accurate input/output protocol implemented in the computation processes.[1]

To realize a functional unit as software, it has to be transformed to pure C++. Although simple applications often run directly on a processor, this work focuses mainly on applications running on an operating systems. As mentioned in Sec. 2.3.2 for DSPs typically RTOS are used. In this case, the basic computation element is the software thread, which corresponds to a software function, which is executed by the OS. The separation of communication and computation can be realized in software as well. Since C++ is fully supported, all facilities for separating design components can be used. The actual computation is modeled very similarly as in the untimed high level model. Only the data types may have to be changed and optimized for the target platform. For example many platforms do not support floating point data types, which have to be transformed to fixed point data types in this case. Synchronization and communication among the threads can be implemented by using shared variables or by specific data structures or facilities like semaphores, which are provided by the operating system. A prominent role for synchronization in software plays the interrupt [Val06, p. 189]. By using interrupts it is possible to synchronize the software execution to external events or to a specific timing.

Table 4.2: Comparison of modeling characteristics of the high level model and of a hardware and software implementation.

	High Level	Hardware	Software
Process type	SC_THREAD	SC_CTHREAD	RTOS specific
Data types	C/C++	SystemC	C/C++
Separation of Comm. & Comp.	Yes	No	Yes
Computation	Untimed	Untimed	Untimed
Communication	SystemC specific	Cycle accurate	RTOS specific

The decisive differences between the high level model and the hardware and software implementation are summarized in Tab. 4.2. Basically all three models realize a functional unit using threads, although the thread type differs in each model. Significant differences are the used data types and the realization of communication, while the actual computation is very similarly implemented. Another important difference is that the separation of communication and computation is not possible in all models. A synthesizable hardware model has to have a pin accurate interface with a cycle accurate I/O protocol.

[1]As mentioned in Sec. 2.2.2, high level synthesis tools provide predesigned interface protocols as well. However, since the restriction to predesigned protocols significantly reduces the design flexibility, the here developed design flow should support user defined protocols as well.

4.1.3 An Adaptable Computation Module

As illustrated in Sec. 4.1.2, modeling differences exist in SystemC between the high level model, the software implementation and the synthesizable hardware model. While communication is modeled significantly different in those three models, computation is implemented untimed in all of them. Although communication and computation are separated into different design components, modeling differences exist, which prohibit the direct use of a high level computation module for hardware synthesis and for software implementation.

One way to overcome the modeling difference is the manual successive refinement of the high level model either to a hardware or a software model. The successive refinement or rather the possibility to model designs at different abstraction levels is a key feature of SystemC. However, it is not suitable for the generation of many different hardware/software partitionings, as it is required for evaluating many different design solutions. Eventually, every design component has to be refined manually to hardware and to software. Therefore, an automated translation from the high level model to either hardware or software is preferable for such a design space exploration.

In this Section, an adaptable computation module is developed, which can be used in the high level model, for hardware and software implementation without changing the module's source code. Therefore, in Sec. 4.1.3.1 a module structure is designed, which enables the transformation from a high level module to a synthesizable hardware module. In Sec. 4.1.3.2 this concept is extended to enable the use of the same high level computation module also for software implementation.

4.1.3.1 Transformation to Hardware

Solutions exist, which enable the automatic translation from a high level model to a synthesizable hardware description. An example has been introduced in Sec. 3.2.1. Haubelt *et al.* present in [HSKM08] the SystemCoDesigner. In Haubelt's work, the high level model has to use the SysteMoC, a model of computation especially for streaming applications. Due to this model of computation, the SystemCoDesigner is limited to data flow applications, which is the main drawback of this solution.

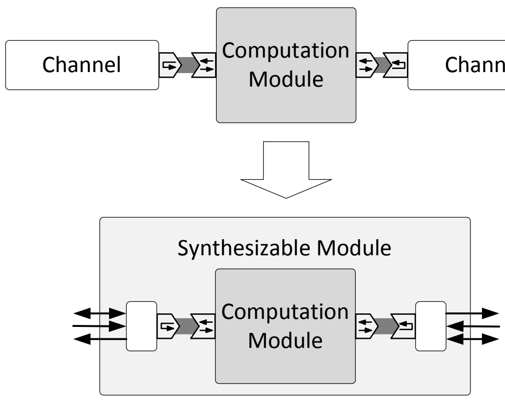


Figure 4.3: The adaptable computation module is connected to adapters, which translate the high level function interface to a cycle and pin accurate signal interface. The adapters and the computation module are instantiated in a top level module, which can be synthesized using a HLS tool.

To avoid such limitations to specific application domains, a different solution is presented in this work. As presented in Sec. 4.1.2, a significant difference between the high level and the hardware model is that synthesis tools require cycle and pin accurate input and output protocols in a synthesizable computation module. This leads to a model, where communication and computation are not separated anymore. Groetker *et al.* present in [GLMS02, p. 162], the adapter concept for hardware communication refinement. The adapter is a design component, which translates the high level functional interface to a low level pin accurate interface. Comparable to a channel it is connected to the port of a module. Each port defines an interface consisting of functions, which have to be implemented by the connected adapter. On the other side, the adapter has pin-level ports, which are accessed by cycle accurate input/output protocols implemented in the adapter's functions. Thereby, a function call from the computation module is translated to a low level, cycle accurate input/output port access. In this way such an adapter can be used to extend a computation module with a cycle accurate input/output protocol. If the high level module and those adapters are connected and instantiated in a top level module, this leads to the basic structure of a synthesizable hardware module, see Fig. 4.3. Thereby, it is possible to generate a synthesizable module structure without changing the source code of the computation module itself.

As Tab. 4.2 in Sec. 4.1.2 shows, the utilization of communication adapters is not sufficient to build a synthesizable module from a high level computation module. Thus, e.g. data types and process types have to be adapted as well. In the following simple coding guidelines and basic C++ features are utilized to design a high level module adaptable. The resulting module can be used in a high level model as well as in a synthesizable hardware model without changing the code itself. An example of such an adaptable module is shown in Lst. 4.1.

```
template <typename TData, typename TAddress, ...>
SC_MODULE(AdaptableModule) {
  HWSW_MODULE                    //-> sc_in<bool> iClk, iReset;

  //other ports ...

  SC_CTOR(AdaptableModule) {
    HWSW_THREAD(Foobar);         //-> SC_CTHREAD(Foobar, iClk.pos());
  }                              //   reset_signal_is(iReset, true);

  void Foobar() {
    {
      ...
      END_RESET;                 //-> wait();
    }
    while(1) {
      ...
    }
  }
};
```

Listing 4.1: The HWSW-Module, an adaptable SystemC computation module. Using template parameters and preprocessor macros a translation from a high level to a hardware module can be realized without changing the module implementation itself. The comments on the right side show the preprocessor transformations for hardware implementation.

For the adaption of data types C++ provides a convenient facility, the template parameters [Sch98, p. 461]. These parameters enable the specification of the actual data types of variables and input and output ports at the time of the module instantiation. Thereby, data types of a high level computation module can be changed throughout the design process. So it is possible to first use high level data types, which are later replaced by refined hardware data types with arbitrary bit width. To gain the most flexibility it would be required to introduce template parameters for each variable in the algorithm. Since many template parameters decrease code readability, a trade off between readability and flexibility has to be found.

Preprocessor facilities can be deployed to adapt the process type and to add clock and reset inputs for hardware synthesis [Sch98, p. 237]. Using the `#define` keyword identifiers can be replaced and via conditional compilation like `#ifdef` it is possible to define different macros for the high level model and for the hardware model. In this way, clock and reset are added for hardware synthesis. Therefore, a new identifier `HWSW_Module` is used in the computation module, see line 3 in Lst. 4.1. Based on this identifier, the adaptable computation module is from now on called HWSW-Module in this work. This new identifier is either removed completely for the high level simulation or it is replaced by a clock and a reset port. In a similar way, the process type can be made adaptable. Instead of using one of the SystemC process types, the `HWSW_THREAD` identifier is used. Via a preprocessor macro, this identifier is either replaced by `SC_THREAD` or `SC_CTHREAD`. For hardware synthesis a function call to `reset_signal_is()` is added as well.

Another identifier is added at the end of the reset initializations, see line 14 in Lst. 4.1. It is replaced by a simple `wait()`, which is required by HLS tools to recognize the reset cycle. For the high level simulation, the identifier can be removed completely or it can be replaced by a `wait()` with a specified wait time. If other `wait()` statements are used in the process to model a realistic execution time, they can be removed in the same manner.

The HWSW-Module is not restricted to a certain application domain. Its adaptability is achieved by utilizing preprocessor macros and template parameters. Together with cycle and pin accurate adapters a synthesizable hardware module can be built without modifying the code of the high level computation module.

4.1.3.2 Transformation to Software

With the adaptable HWSW-Module presented in Sec. 4.1.3.1 it is possible to have a single implementation for computation, which can be used in the high level and in the hardware model. Many of the modeling differences are bridged by the use of adaptations realized with preprocessor macros and template parameters. Equally, the modeling differences between the high level model and the software implementation have to be bridged to get a realization independent model, where different design components can easily be mapped to hardware and software.

A significant difference of the software implementation to the high level model is that it requires a pure C/C++ implementation based on some RTOS. SystemC is a priori not supported. A solution would be to port the SystemC library to the target platform. In this case the complete SystemC standard would be supported for software implementation. Though, the use of the SystemC scheduler atop the operating system would decrease the system performance. Further, each switch to a different RTOS requires a lot of porting effort.

Another solution is presented by Herrera *et al.* in [HPSV03]. In this work, not the complete SystemC library is ported, but a layer is implemented atop the RTOS, which adapts the RTOS application programming interface (API) to the SystemC API. The different SystemC processes are executed as RTOS threads and are scheduled by the RTOS scheduler. Calls to SystemC communication mechanisms are rerouted to RTOS facilities. In this way the whole SystemC standard is supported for the software implementation without having the disadvantages of the solution where the whole library is ported. Obviously, the porting effort is lower compared to the first solution and since the threads are executed directly as RTOS threads, the performance is expected to be higher.

The alternative to adapt the platform or the RTOS to support SystemC, is to adapt the application design itself. Grotker *et al.* in [GLMS02, p. 169] describe the adaption of a SystemC module to a C `struct` using preprocessor macros. Thereby, each SystemC thread is transformed to a POSIX [IG08] thread. In a similar way it is possible to transform the HWSW-Module presented in Sec. 4.1.3 to a pure C++ class. Since a similar approach is used for the transformation to hardware, this offers a consistent solution for this design flow.

```
 1  template <typename TData, typename TAddress, ...>
 2  SC_MODULE(AdaptableModule) {               //-> class AdaptableModule {
 3    HWSW_MODULE                              //-> public:
 4
 5    //other ports ...
 6
 7    SC_CTOR(AdaptableModule) {               //-> void Thread() {
 8      HWSW_THREAD(Foobar);                   //->   Foobar();
 9    }
10
11    void Foobar() {
12      {
13        ...
14        END_RESET;                           // removed
15      }
16      while(1) {
17        ...
18      }
19    }
20  };
```

Listing 4.2: The HWSW-Module with software adaptions. The SystemC module is transformed to a C++ class with two public functions. The comments on the right side show the preprocessor transformations.

To perform this transformation, all SystemC specific macros have to be replaced. The HWSW-Module is shown again in Lst. 4.2. This time the adaptions for the software implementation are added as comments. The `SC_MODULE` is substituted by the key word `class` and `HWSW_MODULE` by `public:`, see line 2 and 3 in Lst. 4.2. Thereby, the SystemC module is transformed to a C++ class.

The SystemC constructor `SC_CTOR` is responsible for registering the threads at the SystemC kernel. In the software implementation, this constructor is not needed anymore and can be replaced by any function e.g. `void Thread()`, see line 7 in Lst. 4.2. The macro `HWSW_THREAD` can be exchanged with a call to the function `Foobar()`, see line 8. If just one thread exists in the module, it can be executed by calling the function `void Thread()`. This leads to a uniform thread name for all modules. If this is not required and the thread `Foobar()` is executed directly, the identifier

`HWSW_THREAD` can be removed entirely. Another remaining nonstandard identifier is `END_RESET`, which is not required for the software implementation and therefore removed as well.

Since typically standard C++ data types are used for the high level model it is often not necessary to refine them for the software implementation. On the other side the data types may have to be changed if e.g. floating point data types are used and the target system only supports fixed point. Equally, if SystemC data types are used, they have to be replaced for the software implementation. In any case, the utilization of template parameters makes the used data types adjustable.

The last remaining SystemC construct in the HWSW-Module is the `sc_port`. To maintain the separation of communication and computation also for the software implementation a similar construct is required. The simplest solution is to use a basic implementation of the `sc_port` class. Thus, the `sc_port` identifier does not have to be replaced in the HWSW-Module and further this enables the connection of modules with communication channels in software.

With the presented adaptions it is possible to transform the high level module to a C++ class, which is compilable without the SystemC library. The decision to use C++ for the software implementation may imply a certain performance overhead compared to pure C. However, it hardly restricts the possible target platforms since today C++ compilers exist even for most DSPs, see Sec. 2.3.2 and it enables the utilization of high level language features. These high level language features increase the abstraction level and thereby simplify the design. A simple design process with a high abstraction level is especially demanded for design space exploration at the system level. Once the best hardware/software partitioning has been found, the actual implementation can be switched to pure C to further optimize the design.

Another performance overhead may be caused by the one-to-one mapping of SystemC threads to RTOS threads. This can lead to many parallel software threads, so that a significant amount of time is spent scheduling these threads. Again, the actual implementation can be optimized in this case by combining different SystemC threads to one RTOS thread. For the design space exploration the one-to-one mapping has the advantage that single components can easily be switched from hardware to software and vice versa. In an optimized thread structure this becomes more difficult.

The presented solution enables a realization independent computation module. Using the shown adaptions, the module can be transformed from a high level module to either a hardware or software implementation. The actual module implementation does not have to be changed. This is only true if the actual computation is synthesizable and compilable at the same time. As presented in Sec. 2.2.3, there are several limitations especially concerning data structures. These limitations are the focus of Sec. 4.1.4.

4.1.4 The Separation of Data Structures

In the previous Sections the HWSW-Module, a realization independent computation module, has been created. Thereby, for the main part the structure of the module has been made adaptable. A SystemC module is converted to a pure C++ class for software design or its ports are extended by a clock and reset port to transform it to a synthesizable hardware module. The thread type is changed from a high level SystemC thread to a clocked hardware thread or to an ordinary C++ function.

The assumption was made that the basic computation can be modeled equally in all three models. However, as already presented in Sec. 2.2.3, high level synthesis tools have specific restrictions and limitations concerning data structures and data types. Thus, a HWSW-Module can only be switched between hardware and software if those restrictions are considered. These limitations prevent the efficient use of complex data structures in the HWSW-Module.

In Sec. 4.1.4.1 restrictions and limitations concerning the use of data structures and data types in the HWSW-Module are presented. An approach to overcome those restrictions and to allow an efficient use of complex data structures in a HWSW-Module is shown in Sec. 4.1.4.2 in terms of the abstract data type concept. This concept is utilized in a way, so that not only communication, but also complex data structures are separated from the computation. The separation of those design components can be realized using polymorphism. The different kinds of polymorphism and their advantages and disadvantages are illustrated in Sec. 4.1.4.3.

4.1.4.1 Limitations of Realization Independent Computation

With certain restrictions it is possible to model computation realization independent. Hence, one model can be used for high level, hardware and software implementation. This is possible, since all three models can be implemented as sequential, untimed software code. However, there are restrictions especially concerning data types and data structures, which complicate the realization independent implementation of computation.

As already mentioned, comparable data types are used for hardware and software implementation. Typically, they support the same operators. However, the used bit width is often different. In hardware a small bit width is desirable to keep the hardware effort as low as possible. On the other side, in software mostly the native processor bit width is used, since a smaller bit width does not lead to better performance. Often the used bit width also differs between the high level model and the software implementation. The high level model is compiled for a host personal computer (PC), which often has a different native bit width compared to the platform used for the software implementation. As a consequence, the algorithm in the HWSW-Module has to be designed bit width independent so that it can be used as high level module, as hardware implementation and as software implementation. An example for a bit width dependent code would be an algorithm which expects an overflow of a variable at a certain point, e.g. to reset a counter.

Another modeling restriction concerns data structures. Simple data structures like arrays and ordinary classes can be used for all three models (high level, hardware and software) without any restrictions. If memory addresses come into play, it becomes more complicated. Memory addresses are utilized by pointers and references. As already outlined in Sec. 2.2.3, most HLS tools only accept pointers and references if they can be resolved at compile time.

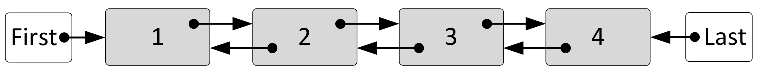


Figure 4.4: The doubly linked list as example for a complex data structure, which heavily utilizes pointers. Further, most software implementations use dynamic memory management to add and remove list elements.

It becomes even more difficult if dynamic memory management is used. In C++ e.g. the standard template library (STL) [56] provides several complex data structures realized with dynamic memory management. A popular example for such a complex data structure is the doubly linked list [SK94], see Fig. 4.4. Each element in the list has two pointers, one to the following and one to the previous element. Two additional pointers indicate the beginning and the end of the list. This illustrates that the pointer is a key element of the doubly linked list.

Especially in high level software and high level models complex data structures are utilized as they significantly simplify the design process. However, in embedded software dynamic memory management is often avoided due to the performance overhead. Commonly they are not used for hardware design, since they are not supported by commercial high level synthesis tools. Reasons therefore are the excessive use of pointers and that dynamic memory management is not supported at all by current HLS tools.

Pointers, references and dynamic memory management assume a linear infinite memory, like it is theoretically available for software. Comparable to the bit width dependent code, an assumption is made about the underlying target architecture. An implementation, which includes such assumptions about the target architecture corresponds to an implementation on a lower abstraction level. The code becomes platform specific, in this case software specific. This makes a realization on another platform, e.g. on an FPGA, more difficult.

As mentioned in Sec. 2.2.3, some solutions have been proposed, which try to synthesize software specific code to hardware structures. The most promising approach by Semeria *et al.* [SSDM01] uses explicit hardware memory allocators to realize dynamic memory management in hardware. These memory allocators require a significant amount of hardware resources. It is arguable whether this additional effort for the realization of dynamic memory management in hardware is worth it. In many cases, a static implementation of such data structures is the more efficient solution. Only under certain premises, the possibility of a static solution without memory allocator is recognized by Semeria's approach.

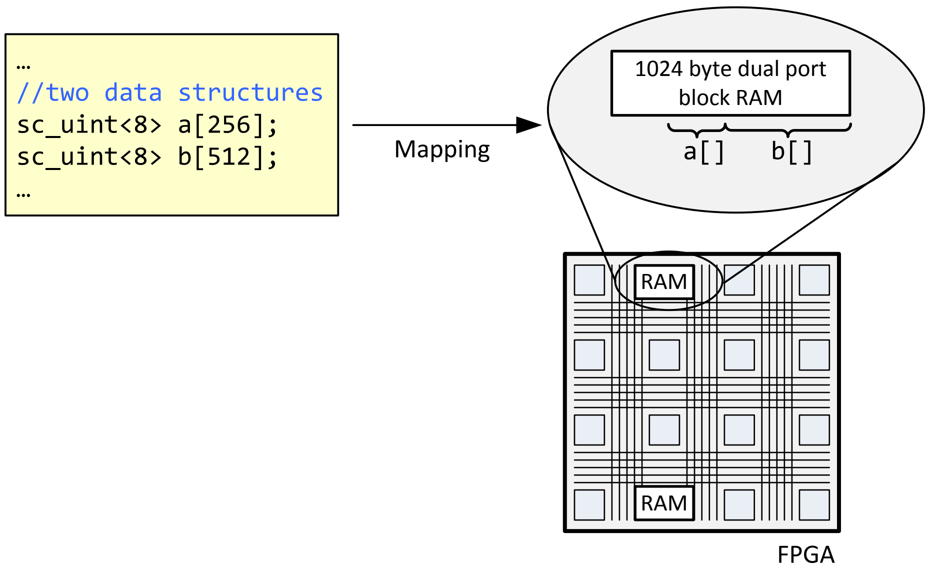

```
…
//two data structures
sc_uint<8> a[256];
sc_uint<8> b[512];
…
```

Figure 4.5: Example of a data structure to memory structure mapping. Two data structures with a size of 256 and 512 bytes are mapped to a 1024 memory structure. Thereby the available RAM block is utilized more efficiently.

The utilization of dynamic memory management in software leads to a design at a lower abstraction level. Equally, the efficient use of hardware memory structures often requires a lower level of abstraction. Especially for FPGA designs it is often necessary to map several smaller data structures to one memory structure to better exploit the limited resources. Fig. 4.5 shows an example where two data structures `a[]` and `b[]` with a size of 256 and 512 bytes are mapped to a 1024 byte block RAM. To realize a data structure to memory structure mapping of this type, some tools, e.g. the HLS tool Cynthesizer from Forte, requires the explicit instantiation of a memory structure module [50]. Due to these reasons, it might be necessary to refine a realization independent computation module after partitioning to efficiently use the available memory structures.

In summary it can be stated that the restrictions concerning data types are negligible. A bit width independent code can be realized using simple coding guidelines. Simple data structures can be used in a HWSW-Module without any limitations as well. However, concerning complex data structures it becomes apparent that different implementations are required for the high level model, the hardware implementation and for the software implementation.

4.1.4.2 The Abstract Data Type Concept

The concept of abstract data types comes from software development. It has been developed as a further abstraction step after the procedural abstraction [LZ74]. The idea is to encapsulate the implementation details of a complex data structure. The basic difference between built-in data structures, e.g. an array, and complex data structures, like a linked list, are the operations, which are provided to modify its content. While the array only provides functions to read and write data, a linked list may provide several functions to add, remove and manipulate elements of the list e.g. by sorting it according to specific criteria. Furthermore, complex and built-in data structures differ in their storage representation. The elements of a linked list for instance may be distributed in the memory, while a simple array is stored linearly.

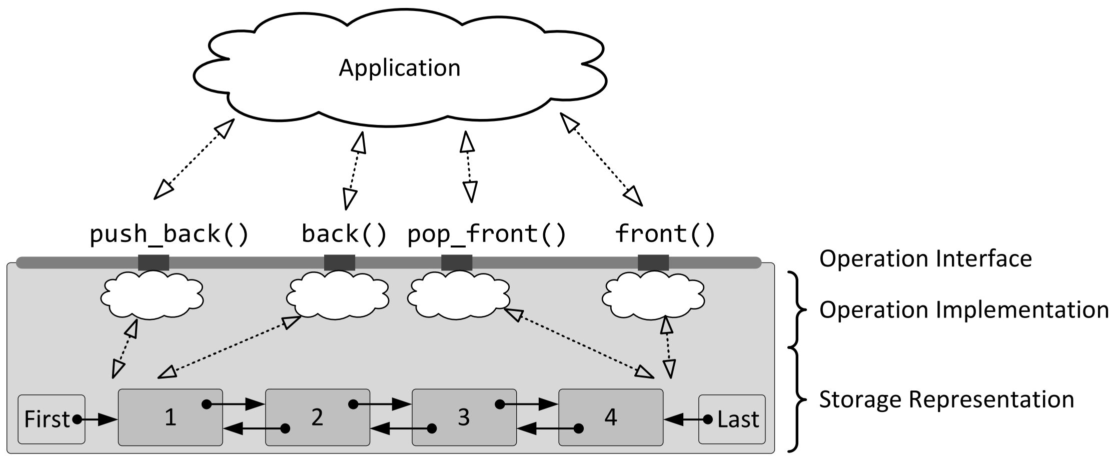


Figure 4.6: Concept of abstract data types. The application only accesses the data structure via the operation interface. Thereby, operation implementation and storage representation are separated from the application implementation.

If the abstract data type concept is used, implementation details like the realization of those access operations and the storage representation are hidden behind a well defined operation interface. This interface works as abstraction barrier. Thereby, the application programmer, which uses the abstract data type, does not have to care about actual implementations nor about storage representations. The actual abstract data type itself does not denote any implementation. It rather denotes a class of implementations, which all provide the same set of operations, hence the same interface.

Fig. 4.6 illustrates the concept of abstract data types. The application program, which uses the complex data structure, only interacts with the operation interface. It solely uses the functionality provided by this interface, but it does not care about the actual implementation. The abstract data type itself is defined by the provided operations. In this example an abstract data type defines a `push_back()` and a `pop_front()` method to add elements at the end and remove elements from the front. Further, via `back()` and `front()` it is possible to read the first and the last element. The example has a simple first-in first-out behavior. An actual implementation can use a doubly linked list as storage representation, see Fig. 4.6. However it might also use a simple linear array to store the different list elements.

A popular example for a library of abstract data types is the already mentioned standard template library (STL) [56]. It actually is a C++ library, which provides so-called container classes and other facilities like iterators. A part of the containers and iterators is today included in the C++ standard library. According to [Sch98, p. 626] a container is an object, which collects internally other objects and which provides operations to add, remove and manipulate objects in the container. Iterators are a generalization of pointers, which allow to access the containers in a pointer-like way. A specific nature of containers is their data type independents. All containers are implemented using template data types. Thus, the data type of the elements, which are actually stored in the container are defined, at the time of the container instantiation. Since different containers provide similar operations, additionally the complexity of each operation is defined. Thus, the user can choose the appropriate container based on the required operations and based on their complexity. Further, this allows a performance estimation of the algorithm using the container without knowing the exact implementation of the container.

As already mentioned, the STL defines different abstract data types without fixing the actual implementation. There are different realizations solely for PCs. For embedded platforms different implementations exist as well. They are optimized for the specific architecture and try to exploit their peculiarities. Hence, the user can implement applications using STL containers and the optimized container implementations ensure a performant realization on the different platforms without the necessity of changing the application code itself.

This idea can be extended across the hardware/software boundary. It can be utilized at the system level. For this purpose, the computation in the HWSW-Module operates solely on interfaces defined by abstract data types without including actual low level data structure implementations. This raises the abstraction level of the HWSW-Module and separates it from low level implementation details. Different implementations can then be used for high level, hardware and software. Low level, realization dependent code structures are encapsulated in the data structure implementation. This reduces design restrictions of the computation module and it enables the use of data structure implementations, which are optimized for the respective target. It is even possible to have different implementations for one and the same target, which are optimized for different purposes.

By utilizing the concept of abstract data types for system level design, it becomes possible to significantly reduce the limitations of realization independent computation modeling. The separation of the application implementation from the data structure implementation allows the use of different optimized implementations in the high level, the hardware and the software model. Thereby, complex data structure can be used in the HWSW-Module and it still can be used in all three models without any changes.

4.1.4.3 Separation Methodology

To utilize the abstract data type concept in the HWSW-Module, a methodology is required to easily exchange the implementation of a data structure. Only then it is possible to keep the module unchanged, while different implementations can be connected for the high level simulation, for hardware or for software design.

In the HWSW-Module all operations are performed on the interface of the abstract data type. Whereby, the interface denotes the different operations the implementation has to support. For each operation its name and parameters are defined. The actual implementation is connected to the interface from outside the module. So that the implementation can be exchanged without modifying the module itself. The basic concept behind separating an interface from the actual implementation is the polymorphism. Literally polymorphism means "having multiple forms". In C++ two types of polymorphism are available [AG04, p. 17]: dynamic and static polymorphism. In the following both kinds of polymorphism and their applicability for separating a data structure interface from its implementation are analyzed in more detail.

Dynamic Polymorphism

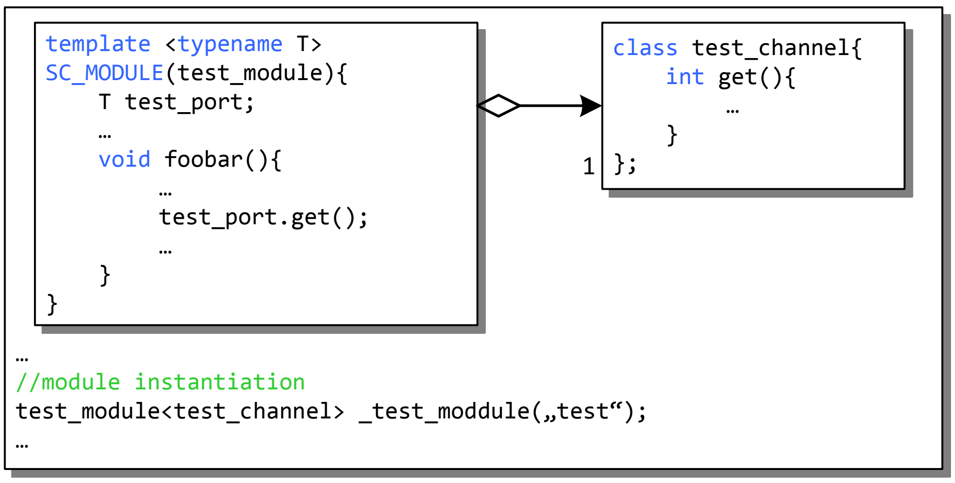
Dynamic polymorphism has been presented in Sec. 2.1.2. In this case, the separation of an interface from its implementation is based on pointers and inheritance which enables the replacement of the implementation even during runtime. It is also possible to connect different modules to one and the same implementation. Both features are not necessary to separate a data structure interface from its implementation. Only a static one-to-one mapping is required. However, the mapping of different data structures to the same memory structure demands such a one-to-many mapping, see Sec. 4.2.2.

A disadvantage of the dynamic polymorphism is the usage of pointers and inheritance. Both are C++ features, which are only partly supported by current HLS tools. For example, the Cynthesizer from ForteDS is not able to resolve multiple hierarchies of pointer connections. In other words, a function called through an `sc_port`, which internally calls a function connected via another `sc_port` is not synthesizable.

Static Polymorphism

Static polymorphism is based on generic programming, hence on template classes. The actual implementation is set via a template parameter. A possibility to separate the interface from the implementation via generic programming is shown in Fig. 4.7. In this case, only two classes are necessary. The `test_module` has a member variable `test_port`, whose type is set via a template parameter. The operations performed on the template parameter implicitly define the interface of the connected implementation. In this example this is a simple `get()` function. The second class `test_channel` denotes the corresponding implementation. The connection between interface and implementation is established in the top level module, when the `test_module` is instantiated. At this point the data type of the member variable is set to the type `test_channel`.

The connection between implementation and interface is resolved already by the compiler and the connection has to be fixed when the components are instantiated. This type of polymorphism does not offer the same design opportunities as dynamic polymorphism. Connecting different modules to the same implementation results in code reproduction. Thus, different objects of the implementation are generated. However, a big advantage of static polymorphism is that it does not require the use of pointers and virtual functions. This results in a better support by HLS tools.



```
template <typename T>
SC_MODULE(test_module){
    T test_port;
    …
    void foobar(){
        …
        test_port.get();
        …
    }
}
```

```
class test_channel{
    int get(){
        …
    }
};
```

```
…
//module instantiation
test_module<test_channel> _test_moddule(„test“);
…
```

Figure 4.7: Example of connecting design components using static polymorphism. The actual implementation of `test_port` is set at the time of the instantiation of `test_module` via a template parameter. The class `test_channel` can be used, since it implements the required `get()` function.

Following these reasons, static polymorphism is used to separate the implementation of a data structure from its interface. Hence, in the HWSW-Module, not only the data types, but also the actual implementation of the data structures is made adaptable via template parameters.

4.1.5 A Tripartite Design Flow

Utilizing the abstract data type concept as presented in Sec. 4.1.4, not only communication but also complex data structures are separated from computation [BHG10]. A functional unit, which is modeled at the system level, has to be divided into computation, communication and data structures.

In this way it is possible to design the pure computation in a module, which is directly synthesizable and compilable at the same time. Hence, except for some limitations, which are presented in Sec. 4.1.6, the module is realization independent. This module called HWSW-Module can be used in the high level, the hardware and the software model. To overcome the modeling differences between these three models, preprocessor macros and template parameters are utilized. Thereby, the module structure and the thread type can be adapted for the respective model.

Communication and synchronization, which are modeled significantly different in the three models, are implemented in channels, which are connected to ports of the computation module. This results in a completely untimed computation module. As presented in Sec. 2.1.2, the type of separation is based on dynamic polymorphism, which might limit the modeling possibilities, since many HLS tools do not support several hierarchies of port to channel connections. However, it is the standard way of realizing the separation of communication and computation in SystemC and the connection of one channel to two modules corresponds to a one-to-many mapping, which can only be realized using dynamic polymorphism, see Sec. 4.1.4.3.

Data structures are separated as well, since an efficient memory usage often requires significantly different data structure implementations for hardware and software. This separation on the one hand concerns complex data structures, which provide particular operations to modify the data structure and which often have specific storage representations. On the other hand this separation also concerns simple data structures like arrays. The reason is that as mentioned in Sec. 4.1.4.1 the efficient utilization of memory structures in FPGAs often requires the mapping of different data structures to one and the same memory structures. To realize such a mapping in hardware, a different data structure implementation for hardware has to be provided. Therefore, especially larger arrays should be separated as well. The separation of data structures from the HWSW-Module is realized by using template parameters, hence static polymorphism.

Data structures, which are used for data exchange between two modules like shared memory take a special position. In general, they can be classified as data structures as well as communication. However, such data structures have to have special facilities for synchronizing the access to the data structure. Further, they have to be connected to more than one module, which is not possible using static polymorphism. Due to these reasons, data structures which are used by more than one module, are classified as communication components.

Both, communication channels and data structure implementations can be replaced with refined, optimized and platform specific implementations without modifying the HWSW-Module itself. As well, it is possible to build libraries for often used data structures and communication components to automate the refinement. An example for a data structure library with different implementations for several widely used data structures is the Hardware/Software Codesign Template Library (CTL), which is presented in Sec. 4.2.

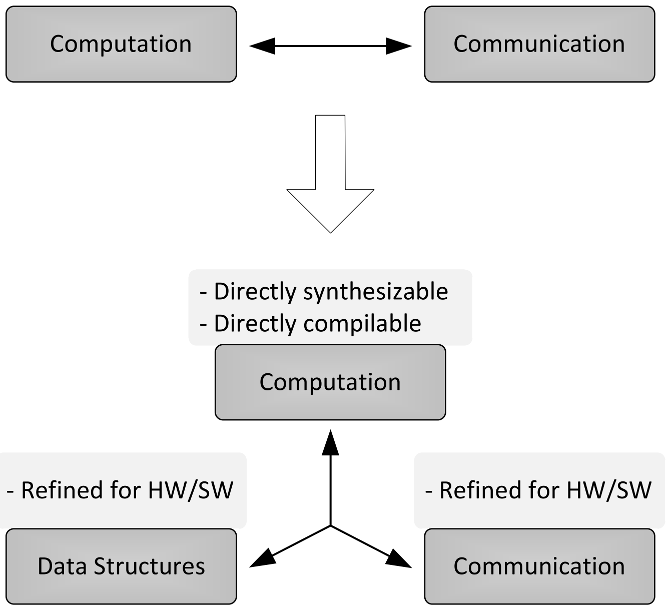


Figure 4.8: By separating not only communication and computation but also data structures, the bipartite modeling paradigm is extended to a Tripartite Design Approach (TDA). Using this separation, the computation can be modeled directly synthesizable and compilable at the same time. Data structures and communication components are refined for hardware or software or they are replaced by library components.

By separating not only communication and computation but also data structures, the bipartite modeling paradigm is extended to a Tripartite Design Approach (TDA), see Fig. 4.8. It enables a realization independent computation module, which can easily be switched between hardware and software. The module is reduced to its pure functionality and does not include target specific code. The exploitation of characteristics of the respective target architecture is ensured by the target specific refinement of communication and data structures. Whereby, communication and data structure libraries can automate this refinement.

4.1.6 Restrictions and Limitations

The tripartite design allows the use of complex data structures in system level models without loosing the flexibility to move design components from hardware to software. This significantly extends the designer's modeling capabilities. However, to ensure a true realization independent computation module, which is directly synthesizable and compilable at the same time, certain restrictions and limitations have to be considered.

Some of those restrictions are caused by the preprocessor macro system, which enables the module adaption for high level simulation, hardware or software design. Only SystemC constructs, which are transformed to compilable software code by a preprocessor macro, can be used in the computation module. An example for such an restriction is the constructor of the HWSW-Module. Currently no transfer parameters are allowed, because this would require the SystemC macro `SC_HAS_PROCESS` for which no adaption macro for software exists. Obviously, such an adaption can be added easily.

Another restriction caused by the adaption system affects the process type. For now, the process type `SC_METHOD` is not supported at all. This corresponds to a restriction to active process types, since `SC_METHOD` is the only way in SystemC to model a passive component. However, as this process type is often only used to model hardware at RTL and as each passive `SC_METHOD` can be transformed into an active `SC_THREAD`, see [GLMS02, p. 169], this should not denote a significant limitation.

SystemC provides several possibilities to model hierarchy and parallelism. Not all of these modeling possibilities are handled by adaption macros. Currently, parallelism is only allowed on the module level. Hence, a module has to have only one thread, it is not allowed to have submodules or multiple parallel threads in one module. In both cases a communication channel may be required to handle the communication among the threads or the modules. To keep such a module adaptable, a possibility has to be provided to replace this channel with a hardware or software refined channel without modifying the module's code.

Other limitations are related to restrictions of the used HLS tool. Although, the Tripartite Design Approach in general has been designed tool independent, each HLS tool has its own limitations and may require additional specific code constructs. The case study, presented in Ch. 5, has been designed using the Cynthesizer from Forte. The Cynthesizer requires specific macros to correctly specify the reset cycle. If not needed, these macros can be simply removed using preprocessor directives. To specify additional design constraints e.g. to unroll a loop or to specify the maximum latency of some part of the design, additional macros are required.

Further restrictions concern all code constructs not supported by the used HLS tool. Although the separation of data structures allows the use of dynamic memory management for the software implementation, it is of course still not possible to use it in the synthesizable computation module to e.g. dynamically generate a variable or even a whole data structure. Also if pointers and references are used, the mentioned restrictions have to be considered. However, with the abstraction of data structures it becomes more unlikely that such low level code constructs even have to be used.

4.2 A Template Library for Hardware/Software Codesign

In Sec. 4.1 the Tripartite Design Approach has been presented. In this approach, not only communication, but also complex data structures are separated from pure computation components. Using the concept of abstract data types [LZ74], realization independent computation components can be built, which can be used unchanged for hardware and for software implementation [BHG11].

In this Section a library is presented, which provides the user with actual data structures for seven different abstract data types. For each abstract data type, data structure implementations for different design phases exist. The system level model is simulated with the high level implementation. During refinement this implementation is replaced by either a hardware specific implementation or by a software implementation. To enable an efficient data structure to memory structure mapping, hardware specific implementations of abstract data types do not actually have internal memory structures. Rather, a memory mapping facility is provided, which enables also the mapping of different data structures to one and the same memory structure. This mapping facility allows the designer to explicitly choose whether to map a data structure to registers or to an FPGA block RAM.

The presented data structure library does not provide data structures, which can be used for communication between different modules or threads. As already explained in Section 4.1 both hardware and software design requires special synchronization mechanisms for such communication data structures, which are not included in the current library elements.

Although many data structure elements of the library are inspired by the C++ standard template library (STL) [56], the intention was not to provide an STL for hardware/software codesign. The main focus was to enable the application of commonly used data structures during all design phases of a system level design space exploration. Since STL containers are used heavily, especially during high level algorithm design, it was a logical step to provide a subset of those containers throughout the whole design process.

The whole library has to work closely with the used high level synthesis tool. Although the larger part of the library is tool independent, some design decisions have been influenced by the supported features of the ForteDS Cynthesizer, which has been used as high level synthesis tool. Obviously, the most tool dependent part is the memory structure mapping feature, which is described in more detail in Section 4.2.2.2.

4.2.1 Elements of the Codesign Template Library

The Codesign Template Library (CTL) currently consists of seven different abstract data types, see Fig. 4.9. The first two elements, Array and Const Array, are simple static data structures. Primarily, they have been added to the library to enable the hardware memory mapping possibility also for such simple data structures. All other data types are dynamic data types, hence their size is not set a priori. For high level simulation and software design, a dynamic implementation is available. This implementation uses dynamic memory management to enable the adaption of the size of the data structure at runtime. The design of these dynamic data types has been influenced by their counterparts of the STL.

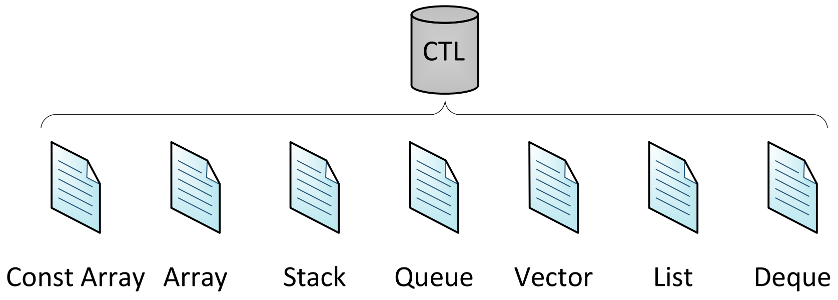


Figure 4.9: Elements of the Codesign Template Library. Currently the library consists of two simple and five dynamic data structures. The Const Array and the Array are mainly added to support the memory mapping facility also for such simple data structures.

At the moment, the library only includes so-called sequence containers. A sequence container is a container, where elements are ordered following a strict linear sequence opposed to e.g. tree data structures [Sch98, p. 626]. Due to this linear ordering, the iteration over the elements can be performed in linear time. Of course the library can and might be extended with other data types in the future. Although such an extension to other container types may increase the applicability of this library, the currently included sequence containers represent a solid basis with commonly used data structures.

Following the concept of abstract data types, the user does not have to know the actual implementation details. Each CTL element has a well-defined interface, which provides operations to add, remove or manipulate data inside the container. Based on the defined interface, every abstract data type is more or less suited for a specific application. Detailed information about the interfaces of all CTL elements can be found in A.1.

Additionally, properties are specified, which define the complexity of the operations. For most of the containers these properties are again following properties defined for STL containers. Due to technical limitations some of these properties have been changed. For example, moving elements or groups of elements between different data structures in hardware cannot be performed as efficient as in pure software, since the use of pointers is heavily limited by current high level synthesis tools. The defined properties help the designer to estimate the algorithm's performance when a specific container is used. In the following all seven abstract data types and their basic properties are presented briefly:

- **Const Array:** The Const Array is the simplest data structure of the library. As its name implies, it can be used for constant arrays. Obviously, it is a simple static data structure. However, the use of this data structure instead of the standard C++ array enables the use of the memory mapping capabilities provided by the CTL. Therefore, during refinement towards a hardware implementation Const Arrays in different threads can be mapped to the same dual port block read-only memory (ROM). The memory mapping capabilities for hardware design are described in more detail in Section 4.2.2.2.

- **Array:** Another simple data structure is the Array, which has mainly been added because of memory mapping possibilities. It basically represents a fixed length array. The Array provides a simple interface with random access read and write functions.

- **Vector:** The Vector is very similar to the Array. The main difference is that the size is not fixed, hence it is a dynamic data structure. Similar to the array, all elements can be randomly accessed by their position in constant time. Only removing and adding elements at the end of the Vector can be performed in constant time. The Vector provides a broad spectrum of access functions, which also includes less performant operations like inserting and erasing elements in the middle of the data structure.

- **Deque:** The Deque, or Double-Ended Queue, is basically an extension to the Vector. Additionally, to constant time insertion and removal of elements at the end, it also provides constant time insertion and removal at the beginning. It is not guaranteed that the elements are stored in contiguous locations, which might increase the access time due to more complex address translation. Nevertheless, random access of individual elements can be performed in constant time.

- **List:** The List is usually implemented as a doubly-linked list. Elements can be added or removed anywhere in the list in constant time. Random access to elements is only possible in linear time by iterating over the elements. Additionally a set of operations is provided. These operations are for example to sort the list according to some order or to invert the order of the list.

- **Stack:** The Stack is a specialized container with very limited access functions. It operates in the last-in first-out (LIFO) context, which means that only the recently added element can be accessed or removed. Thus, also random access to its elements is prohibited.

- **Queue:** The Queue is another specialized container with limited access functions. It only supports element access at the beginning and at the end. Elements are always added at the back and removed from the front. Hence, it operates in the first-in first-out (FIFO) context. Just like the Stack, random access to individual elements is prohibited.

4.2.2 Basic Structure

The basic structure of the CTL is shown in Fig. 4.10. The library is divided into three sub-libraries. In the realization independent computation modules only so-called container adapters are used. For each element of the CTL one container adapter exists. These container adapters basically represent the interface, which is later connected to an actual implementation. Based on the better support of static polymorphism by HLS tools, as presented in Section 4.1.4.3, it is used to separate the container adapter from the actual data structure implementation. Accordingly, the container adapter has an implementation as member variable and the actual type of the implementation is set via a template parameter. Fig. 4.11(a) shows the usage of the CTL in a computation component. A more detailed description of the container adapter sub-library can be found in Section 4.2.2.1.

Throughout the different design phases, the container adapter can be connected to different implementations, which are summarized in the container implementations sub-library, see Section 4.2.2.2. Currently, there are implementations for the high level simulation, for hardware and for software realization. Hardware specific container implementations require the consideration of memory structures, which are available in the target technology. To avoid multiple implementations, the data structure implementation is separated from the actual memory structure. Thereby, it is additionally possible to map different data structures to one and the same memory structure. This separation is realized via dynamic polymorphism using `sc_ports`. The static polymorphism solution presented in Section 4.1.4.3 cannot be used, since this solution does not support the mapping of different data structures to the same memory structure. Fig. 4.11(b) shows a memory structure connected to a container implementation using an `sc_port`.

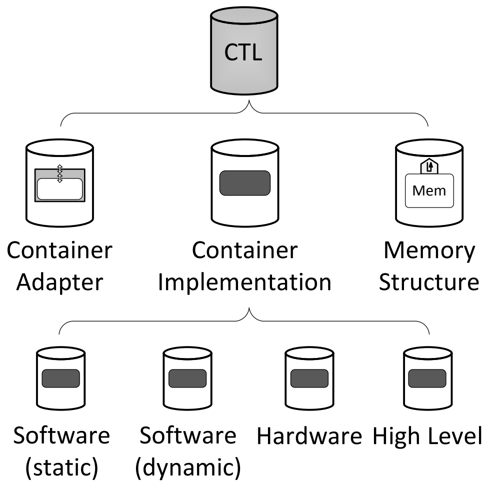
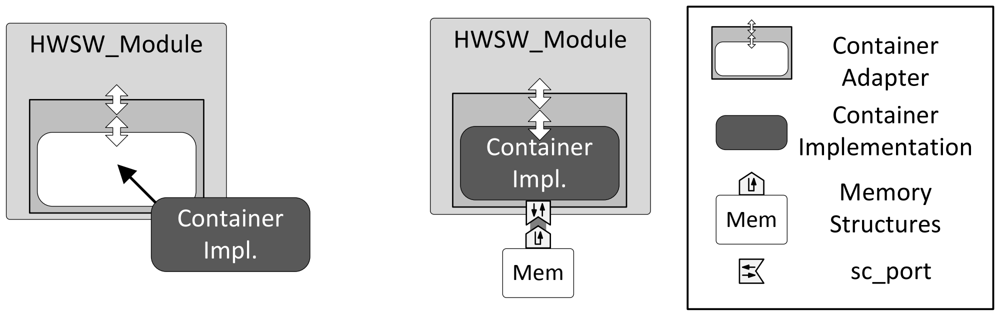


Figure 4.10: Basic structure of the Codesign Template Library. It consists of three sub-libraries: container adapters (data structure interfaces), container implementations (implementations for software and hardware design and high level simulation), and memory structures (specifically for hardware design).



(a) Container adapter connected to an Implementation

(b) HW implementation connected to a memory structure

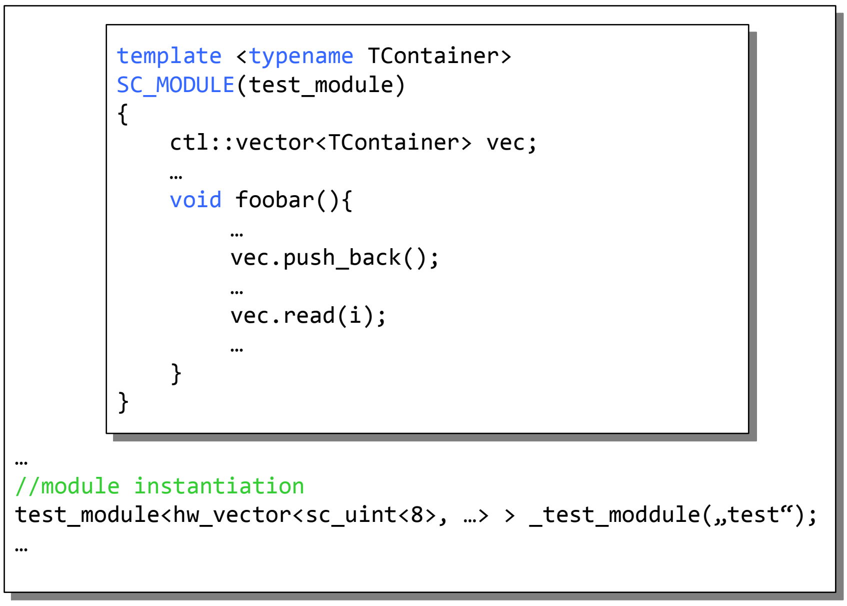
Figure 4.11: Basic functionality of the CTL. In the HWSW-Module, one of the container adapters is instantiated. Fig. 4.11(a) shows the connections of one of the available container implementations to the container adapter. All hardware implementations possess an `sc_port` to whom a memory structure has to be connected, see 4.11(b).

The final sub-library, the Memory Structures sub-library, see Sec. 4.2.2.3, consists of several different memory structures, to which a data structure is mapped if it is realized as hardware. The present library focuses especially on memory structures available in modern FPGAs.

4.2.2.1 Container Adapters

As already mentioned, the container adapters are the elements, which are actually instantiated in the realization independent computation module. For each element of the CTL one container adapter is available. The term container adapter is also derived from the STL. According to [Sch98, p. 629] a container adapter holds internally a container and provides an adapted set of operations to modify the container.

Basically, the container adapter provides the interface defined for the respective abstract data type. Most of the functions simply call the corresponding function of the connected implementation. Only operations, which can be implemented realization independent by using other member functions, are realized in the container adapter itself. In the current set of library elements only the assignment operator, to assign one container to another, and the swap function, which exchanges the content of two containers, are implemented directly in the container adapter.



```
template <typename TContainer>
SC_MODULE(test_module)
{
    ctl::vector<TContainer> vec;
    …
    void foobar(){
        …
        vec.push_back();
        …
        vec.read(i);
        …
    }
}
```

```
…
//module instantiation
test_module<hw_vector<sc_uint<8>, …> > _test_moddule(„test“);
…
```

Figure 4.12: Code example for the usage of a container adapter. The `test_module` instantiates the container adapter Vector. When the module is instantiated, the template parameter `TContainer` is set so that the hardware implementation is used.

Fig. 4.12 shows an example for a computation module named `test_module`. In this module, the library element Vector is instantiated. To avoid name conflicts, all container adapters of the CTL are collected in the namespace `ctl`. Each container adapter has at least one template parameter: `TContainer`. This parameter represents the container implementation. When the computation module is instantiated, the actual implementation is specified. In this example the hardware specific implementation of the Vector is used.

Every implementation connected to a container adapter has to define two data types: one named `data_type`, which is the data type of the data stored in the container and one named `size_type`. The second one is used for size and address values. Depending on these data types defined in the implementation, the container adapter sets its own data types. This is shown e.g. in line 5 and 6 in Lst. A.1. This ensures that the adapter works with the same data type as the connected implementation. Especially for efficient hardware design with minimum resource usage it is necessary to use data types with arbitrary bit width also for the data type of size and address values.

As mentioned before, some containers are influenced by STL containers. Consequently, also the interfaces are closely to the interfaces of the STL containers. However, technical limitations, especially resulting from limitations from the HLS tool, lead to slightly modified interfaces. For example, the Cynthesizer from ForteDS does not allow the use of references or pointers to a dual port block RAM. Hence, the square bracket operator, which usually returns a reference to a single memory cell, has been replaced by a read and a write function, see e.g. line 22 and 23 in Lst. A.3. Iterators, which are basically pointers are also not supported. Only the List requires an iterator-like construct to iterate efficiently through the elements. Therefore, a iterator solution, which does not require the use of pointers has been implemented. Details thereto can be found in Section 4.2.3. The operations defined in the interface can be divided into different categories. Each category is described shortly in the following:

- **Capacity:** Each container provides different operations to determine information concerning the size like the current size or the maximum size. Some containers also provide functions to change the size of the container to a specific value.
- **Element Access:** The element access functions enable reading or writing of different elements of the container. Examples are `front()` and `back()`, which simply read the first and the last element of the container or `read()` and `write()`, which allow reading and writing of elements selected via an address.
- **Modifiers:** Modifier functions provide the possibility to modify the content of the container. In contrast to the element access function, modifier functions allow to change the size by adding or removing elements.
- **Operations:** The last category, the operations, are special functions, which are at the moment only provided by the List. These functions allow to e.g. remove elements, which have a specific value or to change the order of the elements.

4.2.2.2 Container Implementations

The second sub-library of the CTL contains different implementations for each library element. In general a rich variety of container implementations is possible. By providing different implementations for various applications and target technologies, it is possible to account for the differences in their memory architectures. The presented implementations are just examples, which demonstrate the concept and its usability for hardware/software codesign and design space exploration. For each container four different implementations have been designed: a high level implementation for system level simulations, a synthesizable static hardware implementation and two software implementation.

High Level Implementations

The high level containers are intended for the system level simulation. Hence, one requirement is a fast implementation to ensure high simulation speed. For that reason the utilization of optimized STL elements provided by the C++ Standard Library [ISO03] suggests itself.

Both for hardware and for software, static implementations are available. To perform the switch from a dynamic high level implementation to a static implementation, the maximum memory requirements have to be estimated. Therefore, a profiling feature has been added. It logs the maximum memory utilization during an application. By simulating realistic application scenarios an estimate of the required memory of each container can be generated. The accuracy of the estimate obviously relies mainly on the simulated scenarios. Lst. A.8 shows a code snippet of the Vector's high level implementation.

As already mentioned, each implementation has to define two data types. The data type of the stored data is set via a template parameter. The type used for size and address values is derived from the used container of the Standard Library. The Array has an additional parameter to specify the size. Internally, the container instantiates a common C++ array. The Const Array container does not need a size parameter. It basically consists of a pointer to a constant array, which has to be defined outside the container implementation. The address of the array is assign to the pointer via a function. The size of the container is solely defined by the size of the constant array.

Hardware Implementations

The hardware implementation should provide an efficient and synthesizable implementation of the CTL elements. Although a draft of a synthesizable subset of SystemC is available [OSC09], differences between certain HLS tools exist. Thus, it has been decided to optimize the implementations for one chosen HLS tool, which is in this case the Cynthesizer from ForteDS. The optimization results in a specific subset of SystemC and in design constraints, which are added in the source code. Lst. A.9 shows a code snippet of the Vector's hardware implementation.

All hardware containers have certain similarities. First of all, each hardware container is basically only a container adapter, since the actual structure where the data is stored is separated in the memory structures. Therefore, every container has an `sc_port`, to which the used memory structure is connected. In the container, the complex interface functions of the abstract data type are reduced to simple read and write operations performed on the port. When the port is mapped to a memory structure, an additional parameter can be used to define an address offset, which is added to the address of each read and write function call. Thereby, it is possible to map different hardware containers to different address ranges of one and the same memory structure.

Another similarity are the template parameters. Every hardware container has three template parameters. One parameter to set the payload data type, one to specify the bit width of the data type used for size and address values and one parameter to specify the size of the container. All containers, even the dynamic data types, are realized using static memory management. The memory utilization estimation gained during high level simulation can now be used to specify a static container size. Obviously, also a dynamic hardware realization using a hardware memory allocator is possible, see [SSDM01]. However, a dynamic solution would require significantly more hardware resources and in many cases, a static implementation is the more efficient solution.

One syntax restriction, which results from the HLS tools is limited support of pointers. The Cynthesizer does not support pointer arithmetic, several hierarchies of pointers and as already mentioned, it is not possible to define a pointer to a dual port memory structure. Following these restrictions, the implementation of the hardware containers avoids pointers whenever possible. However, some realizations require the use of pointers, which points e.g. to the beginning or the end of the data structure. No matter how the internal structure of the container looks like, the actual memory structure is a single chunk of linear memory with a continuous address space. The address range used by the container results from the offset and the size. The offset, which is set, when the container is mapped to a memory structure represents the beginning of the address range. The end results from the size of the container.

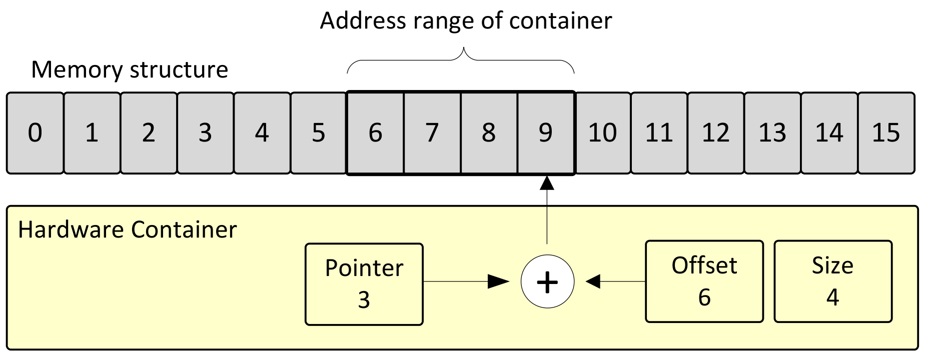


Figure 4.13: Container to memory structure mapping with offset. A container with a size of four is mapped to a memory structure with 16 memory cells. With an offset of six, the container allocates the cells six to nine.

Fig. 4.13 shows a hardware container and the connected memory structure. The memory structure consists of 16 memory cells. The container with an offset of six and a size of four is mapped to the cells six to nine. Based on the known address range, pointers used within the container can be realized without using C/C++ pointer syntax. They can be implemented with simple variables, which are used to store the address of the corresponding memory cell. In this example, the pointer points to the container's last memory cell.

The hardware implementations of the CTL containers basically use three different data structures: array, circular buffer, and doubly linked list. Tab. 4.3 shows the corresponding data structure of each CTL element. The simplest data structure, the array, is used by the following CTL elements: Array, Const Array, Vector and Stack. These containers store the values in a linear sequence. Stack and Vector additionally store the current size of the container. As already mentioned in Sec. 4.2.1, elements can be added and removed at the end of the Stack or the Vector in linear time. The Vector also provides more complex functions to insert or remove elements in the middle or at the beginning of the container. To insert a value at the beginning of the container, all elements have to be moved to the next higher address.

Table 4.3: CTL elements and their basic hardware realization. All in all three different data structures are used: array, circular buffer, and doubly linked list.

CTL Element	Hardware Realization
Array	Array
Const Array	Array
Vector	Array
Stack	Array
Deque	Circular buffer
Queue	Circular buffer
List	Doubly linked list

The circular buffer, which is used for the Deque and the Queue implementation, allows adding and removing elements at both ends without moving other elements [HR94, p. 418]. In the circular buffer, the beginning is not always at the first address. Beginning and end are defined by two pointers. Between these two pointers, the elements are arranged in a linear sequence. The name circular buffer derives from the fact that after the last address in the address range, the first address is written again. If both pointers point to the same address, the container is either full or empty. To distinguish between these two states, the size of the container is stored too.

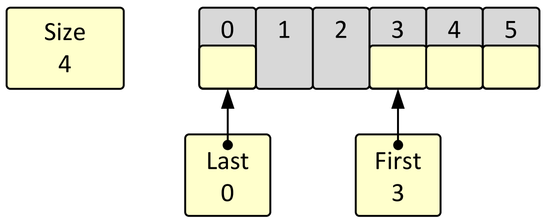


Figure 4.14: Structure of a circular buffer. Four of the six available memory cells are currently occupied by elements. Beginning and end of the buffer are defined by a pointer. Since moving the position of both the beginning and the end is allowed, it is possible to efficiently add and remove elements on both ends.

Fig. 4.14 shows a memory structure with six memory cells. Four cells are occupied by elements of a circular buffer. The illustrated state can be reached after inserting and removing several elements. Thereby, beginning and end of the buffer are moved within the available address range. In the current state, the first element is stored in address three, while the last element is stored in address zero.

The third data structure, the doubly linked list is used to realize the List container [SK94]. An example to realize a doubly linked list in hardware is presented in [XDS$^+$08]. However, the presented solution describes a pure hardware realization. The here presented container implementation is a model at a higher abstraction level. The implementation has pointers to the first and to the last element of the list. The elements do not have to be stored in a linear sequence. Instead, each element has a pointer to the previous and to the next element. This explains that the List does not support random access of elements by their position. It is only possible to iterate over the elements. Therefore, an iterator is provided for the List container, which is explained in more detail in Sec. 4.2.3.

Since the List has to be stored in a linear memory structure, some kind of memory management has to be implemented. Thereto, a status register is provided, which stores the state of each address of the memory structure. If an element is deleted from the List, only the pointers of the previous and the next element have to be adapted. Then the bit in the status register corresponding to the deleted address is set to false, which indicates that the address is free. Currently, the value and the pointers of an List element are converted to a single value with a larger bit width. Thus, a List can only be mapped to a memory structure with a bit width, which equals the sum of the data bit width and twice the address bit width.

Fig. 4.15 shows again a memory structure with six memory cells. This time a doubly linked list with three elements is stored. Whereby, one element is stored at address zero, one at address three and one at address five. Accordingly, in the status register are the corresponding bits zero, four and six set to one, to indicate an occupied memory address.

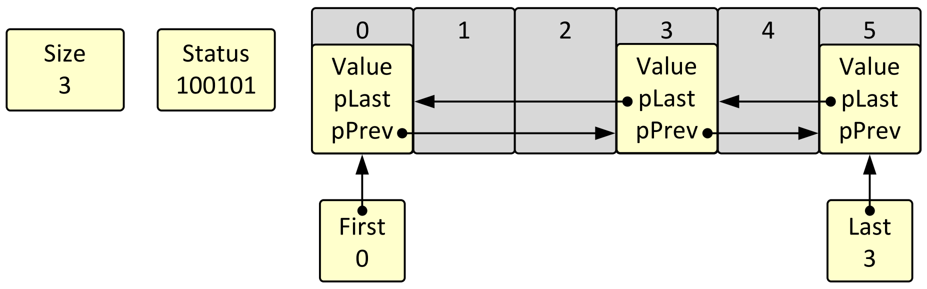


Figure 4.15: Structure of a doubly linked list. Elements do not have to be stored in consecutive memory cells. This enables the possibility to add and remove elements from anywhere within the list without removing all other elements. The "Status" variable keeps track of free and occupied memory cells.

Software Implementations

The software implementations are used if the computation module should be realized on any processor. Two different types of implementations for software are currently available. The first implementations are comparable to the high level container implementations, except for the profiling feature, which has been removed. Besides that, these implementations are pure C++ implementations and they use the containers of the C++ Standard Library. They can be compiled directly by any standard C++ compiler for any given platform. Like in the high level containers, the Array is realized using a C++ array and the Const Array possess a pointer which points to an constant C++ array defined outside the container implementation.

Due to the overhead of dynamic memory management a static implementation might be valuable especially for the design of systems with tight constraints. Therefore, a second sub-library of container implementations is provided. This sub-library consists of static container implementations, whose basic structure is comparable to the hardware implementations. They also utilize data structures like a ring buffer or a doubly linked list. The container size can be derived from the memory utilization estimation gained during the high level simulation. The main difference to hardware containers is that the software implementations do not have a memory mapping feature and they do not include any SystemC constructs.

4.2.2.3 Memory Structures

The third sub-library provides memory structures to which the hardware containers are mapped. In this work memory structures are defined as a specific way of organizing memory and the access to it in hardware. Therefore, a memory structure denotes how memory is organized and how it can be accessed. There are different ways of organizing memory. In software the designer has an abstracted view of the system and usually does not have to care about the specific memory structure. This is different for hardware designs. At least for specific memory structures, many high level synthesis tools require the explicit consideration.

As already mentioned, it is possible to map different containers to one and the same memory structure by providing an address offset. This helps to efficiently utilize memory structures which are available on the target architecture. Currently, the provided memory structures focus mainly on the target FPGA. The library provides elements for common on chip memory structures. However, the library can easily be extended for other targets.

Modern high level synthesis tools provide possibilities to use registers as well as block RAMs. Forte's Cynthesizer, the HLS tool the CTL is optimized for, usually generates VHDL/Verilog code for a block RAM if an array is used in SystemC. Via a synthesis directive, which is added in the code, the tools can be forced to flatten the array to distributed registers. To utilize two ports of a block RAM, it is required to generate a memory module with the Cynthesizer. This module has to be instantiated explicitly in the SystemC code, which makes the code tool dependent. As already mentioned, it is not possible to define pointers or references to such a memory module.

The memory structure sub-library provides different classes, which can be connected to one or more container implementations. With each class a different type of FPGA memory can be utilized. Currently, the library provides six memory structures, which cover the typical on-chip memory types of an FPGA. The memory structures named Ram and Rom use internally a simple C++ array. With an additional synthesis directive, it is ensured that the HLS tool flattens the memory structure into distributed registers. Bram and Brom use a C++ array as well. In this case no directive is required. The HLS tool generates a RAM or a ROM built out of a block RAM. The last two memory structures Dpbram and Dpbrom have two implementations, which are distinguished via a preprocessor directive. If the memory structure is synthesized by the Cynthesizer, a tool specific memory module with two ports is instantiated. During an ordinary SystemC simulation, the second implementation, which consists basically of a simple C++ array, is used. The differentiation is necessary, since the simulation of a Cynthesizer specific memory module requires the complete ForteDS simulation environment.

By using the presented memory structures and the CTL's mapping facility, it is possible to map different containers to the same memory structure. As long as only containers used by one and the same thread are mapped to a memory structure, the HLS tool schedules the memory accesses. The synthesis tool thereby prohibits memory access conflicts. If a dual port memory structure is used, it is even possible to map containers used by two different threads to the two ports of the memory structure. A conflict can only occur if two containers of different threads try to write to the same address in the same clock cycle. Since containers are usually mapped to different address ranges, this conflict can be avoided. Additional logic for scheduling memory accesses is required if containers of more than two threads are mapped to a dual-port block RAM or if containers of more than one thread are mapped to a single-port block RAM.

4.2.3 Iterators in the Codesign Template Library

The STL not only consists of containers, also functors, algorithms and iterators are part of the concept. The general idea of the CTL was to provide complex data structures, which can be used easily for HW/SW codesign. Containers, which support the tripartite system design flow. It is possible to extend the design flow to support more STL concepts. The previous Sections have shown that the limiting factor are the HLS tools and their limitations concerning high level programming language features. Due to the limited support of pointers it is difficult to adapt STL concepts for HW/SW codesign. However, especially containers like the List require a possibility to efficiently iterate over the elements. In [Sch98, p. 628] an iterator is defined as an object, which provides operations to access and manipulate the objects of a container, comparable to the operations provided by a pointer operating on an array.

To realize such a concept, so that it is compilable as well as synthesizable, it is necessary to avoid the usage of pointers as much as possible. In the following a solution is presented, which does not need any real C++ pointers. The functionality is of course limited compared to the pure software iterators. However, the basic concept stays the same. Different from the STL where various iterator types are available, only one simple iterator type is provided at the moment.

```
template <typename TContainer>
SC_MODULE(test_module){
    ctl::list<TContainer> _list;
    ctl::list<TContainer>::iterator _iter;
    …
    void foobar(){
         …
         _list.read(_iter);
         _list.increment(_iter);
         …
    }
}
…
//module instantiation
test_module<hw_list<sc_int<8>, 4, 16> > _test_moddule(„test“);
…
```

Figure 4.16: Example module utilizing the List container and an iterator. The actual implementation of the container and the iterator is set via one template parameter. Due to the avoidance of pointers, some iterator functions have to be implemented in the container.

The presented solution has to fit into the existing concept of the CTL. Following the structure of the library, the iterator realization is separated into iterator adapter and iterator implementation. Like the container adapter, the iterator adapter is instantiated in the computation module. Fig. 4.16 shows an example module named `test_module` which instantiates a List and an iterator. One template parameter is used to specify the actual container implementation. The iterator adapter internally instantiates the iterator implementation corresponding to the specified container implementation. In Fig. 4.16 the hardware implementation of the List is selected.

High level and software implementation use the iterator of the C++ Standard Library. The hardware implementation has to realize the iterator concept without using actual pointers. Instead of a pointer, the iterator stores the address of the list element, to which it points, in a simple variable. This address corresponds to the storing location of the list element in the actual memory structure.

By completely avoiding the use of pointers, some iterator functions have to be realized differently than in software. This does not affect simple assignments or compare operations, which are simply overloaded in the iterator class. More complicated are functions like increment, decrement and dereferencing the iterator. Since these are functions, which actually modify the container a pointer to the container is required. To avoid pointer utilization these functions are implemented directly in the container class. The iterator is than passed as a function parameter and is modified. An increment function for example is in the high level and software implementation realized by simply applying the increment operator of the iterator. In hardware, the address stored in the iterator is used to read the corresponding List element. Since each List element has a pointer to the next and to the previous element, cf. Sec. 4.2.2.2, it is possible to update the iterator, so that it points to the next List element. An example is shown in Fig. 4.16. First, the iterator is dereferenced by calling the function `read()`. Then, it is incremented via the `increment()` function. Lst. A.10 shows the implementation of the `read()` and `increment()` function.

In this way different realizations of iterators are summarized in an simplified but realization independent version of iterators at the system level. The workaround that certain functions of the iterator are implemented in the containers avoids the use of C++ syntax pointers.

4.2.4 Restrictions and Limitations

The presented hardware/software Codesign Template Library provides data type independent containers in SystemC. Due to the separation of container adapter and actual implementation, the designer can use a container adapter of a complex data structure at the system level. During the different design phases different implementations can be connected to the container adapter. At the moment four different implementations are provided. Several more implementations can be added in the future. It is even possible to provide dynamic hardware implementations by using a hardware memory allocator.

The CTL concept raises the abstraction level of the system level model and enables the use of complex data structures during system level design space exploration. Currently the library consists of seven commonly used containers. It can easily be extended with even more complex data structures. As mentioned in Sec. 4.2.3, it is also possible to try to integrate other parts of the STL concept like functors and algorithms. However, whether a suitable extension of the CTL can be found depends heavily on the restrictions of HLS tools. With the current limitation concerning pointers and references it will be difficult to find an efficient and convenient solution.

Additionally, a mapping facility for hardware memory structures has been presented. It enables the possibility to map different containers to one and the same memory structure. This provides an efficient way of utilizing available memory structures. Currently, only containers of one thread can be mapped to the same memory port. This limitation can be extended if a memory arbiter is provided, which synchronizes the memory accesses from different threads. At the same time, this begs the question if the resource savings due to the merged memory structures compensate the additional effort to arbitrate the memory accesses. At the moment, the library only provides typical on-chip memory structures of FPGAs. Certainly, the addition of structures for external memories and also for completely other targets would be interesting.

Although the CTL provides the basic features of containers and iterators without using pointers, it is not possible to provide the same functionality as the STL. This especially concerns the data exchange between containers. Since different data structures can be mapped to different memory structures, the data exchange between them can only be realized by copying the data. In software, a continuous memory space exists. Thus, data exchange between containers can be implemented efficiently by only modifying a few pointers.

The mapping facility itself can also be extended. At the moment, different containers can only be mapped to different address regions. However it is possible to add a feature, which conveniently enables the mapping of two containers with e.g. 8 bits x 128 words to a memory structure with 16 bits x 128 words. A comparable functionality is integrated in HLS tool C-To-Silicon [33].

4.3 System Design using the Tripartite Approach

In Sec. 4.1 a tripartite modeling approach for the design space exploration of low volume embedded systems has been presented. The basic idea of this approach is to separate not only communication and computation but also data structures. This tripartite separation enables the realization independent design of computation modules. Hence, one and the same module can be used for hardware and software design. For communication and data structures a library based approach is applied. In Sec. 4.2 a corresponding data structure library has been presented. The CTL provides seven different containers and various different container implementations optimized for high level simulation, hardware and software implementation. A methodology is provided to easily exchange the different implementations. The Tripartite Design Approach combined with the CTL significantly simplifies the generation of several different hardware/software partitionings for design space exploration.

In this Section the application of the TDA concept and the CTL for the design space exploration of a system is presented in more detail. Therefore, Sec. 4.3.1 illustrates the basic design flow of the TDA. The remaining Sections provide further details of different design steps as the system modeling, refinement and evaluation.

4.3.1 Basic Design Flow

In this Section the basic system design flow utilizing the Tripartite Design Approach and the CTL for design space exploration and hardware/software codesign is illustrated. An overview is given of the different design phases and the deployment of TDA and CTL [BHG10].

In a typical board-level system design both a rough architecture and a principle hardware/software partitioning is given. Based on the designers experience and on the given system specification the required processing power can be estimated. Thereof, a coarse system architecture is designed. Often the architecture decision is also influenced by previous design projects. Usually processing platforms, which already have been used, are preferred because of the gained design knowhow. Furthermore, the design typically can be separated into a control flow dominated part and a data flow dominated part. Control oriented tasks are commonly implemented on general purpose processors. Data flow oriented parts can be realized in software using optimized processors like DSPs or they can be implemented as custom hardware blocks on FPGAs. To simplify this difficult and crucial partitioning decision the TDA can be applied. As far as the data flow dominated design components can be isolated from the rest of the design, it is possible to solely model these components using the tripartite approach.

The basic design flow of the design space exploration for these data flow dominated design components is illustrated in Fig. 4.17. According to the TDA, all included tasks are first separated into data structures, computation and communication. The computation is modeled realization independent by using the presented HWSW-Module, which can easily be adapted for hardware or software implementation. All communication and synchronization is separated in so-called channels. Also data structures are separated from the computation. In many cases, elements of the CTL can be applied. This has the advantage that the CTL already provides several different optimized implementations and a mechanism to exchange them easily. In Sec. 4.3.2 the design step of modeling components at the system level utilizing the TDA is presented in more detail.

The system model is then translated to a refined system model, see Fig. 4.17. This step denotes the actual HW/SW partitioning. Each component is either mapped to hardware or software. For design space exploration several different partitionings have to be generated and evaluated. The utilization of the TDA and the CTL reduces the design effort of this step to a minimum. The HWSW-Modules can be transformed to hardware or software implementations by using adaption mechanisms like preprocessor macros or template parameters. The actual implementation code does not have to be changed at all. Communication, synchronization and data structures are refined to hardware or software specific components. In both cases a library based approach is applied. Thus, refinement often only denotes the replacement of high level components by refined components from the library. Only application specific parts, which are not included in the library, have to be refined manually. The refinement step is illustrated more precisely in Sec. 4.3.3.

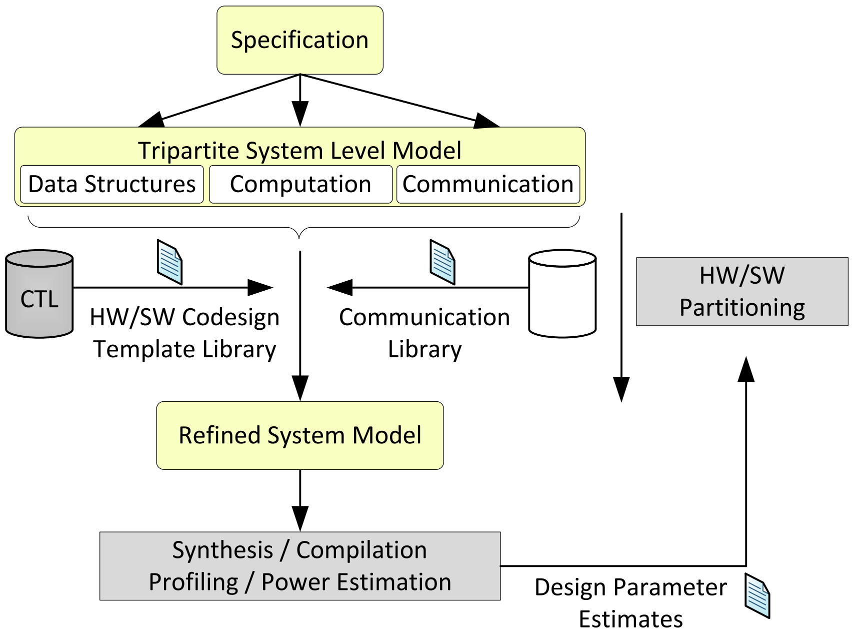


Figure 4.17: Design flow using the tripartite system level design approach. The tripartite separation together with a data structure and a communication library enable the simple generation of different synthesizable and compilable hardware/software partitionings. This further simplifies the estimation of design parameters.

Then, the different generated design partitionings have to be evaluated. Design parameters have to be estimated to classify the solutions in the design space and to verify the fulfillment of design constraints. All design components are either directly compilable or synthesizable depending on whether they have been mapped to hardware or software. Both synthesis and compilation provide useful design parameter estimates. Further parameters can be estimated using low level tools, simulation and prototyping. The generated design parameter estimates support the designer to find the best hardware/software partitioning. The evaluation of the different design solutions is shown in more detail in Sec. 4.3.4.

4.3.2 Tripartite System Level Modeling

To generate and evaluate different application-to-architecture mappings of the data flow dominated part of a system, design space exploration is performed. The tripartite system modeling approach supports the designer during this step. In this Section the design of a first system model utilizing the TDA is described in more detail. This first model is purely functional and it forms the basis for the subsequent design space exploration.

According to the tripartite modeling approach, each system task is separated into computation, communication and data structures. The computation is realized in untimed, realization independent HWSW-Modules. Whereby, each module consist of one thread, in which the actual computation is implemented. The realization independence is enabled by using template parameters to allow the adaption of data types. Furthermore, preprocessor macros are utilized to add or remove certain keywords which are only required either for the high level model, the hardware or the software implementation.

CTL elements are used to implement data structures. In the HWSW-Module solely the container adapter is instantiated. The actual container implementation is connected by using static polymorphism. The CTL provides seven different complex and simple data structures. If the application requires another container with different characteristics, it can be added to the library following the same design concept. For the first functional simulation, the high level container implementations are connected to the adapters.

```
template <typename TData>
class Serial_Read_IF: public virtual sc_interface {
public:
  virtual TData read() = 0;
};

template <typename TData>
class Serial_Write_IF: public virtual sc_interface {
public:
  virtual void write(TData &) = 0;
};
```

Listing 4.3: Basic channel interfaces for serial data exchange.

All communication and synchronization is separated in channels. They are connected to the HWSW-Module via ports and interfaces. A detailed description of the port-interface-channel concept can be found in Sec. 2.1.2. In principle, the designer can choose any interface. However, to ensure a certain interoperability between channels and modules, two simple interfaces are defined, see Lst. 4.3. The definition of standard interfaces increases reusability of of all design components. The defined interfaces called `Serial_Write_IF` and `Serial_Read_IF` provide basic functions for the exchange of single data values. For most data flow components these interfaces are sufficient. Anyhow, user defined interfaces can be added easily.

Data flow dominated designs are commonly modeled by using a process based MoC, see Sec. 2.1.1. Therefore, communication usually denotes the data exchange between two communicating processes. If the whole design can be modeled in this way, a simple set of point-to-point communication channels like FIFOs can be used. However, this is typically only the case if the system is modeled at a very high abstraction level. Often, timing and events become important as the system is refined towards an actual implementation. Therefore, the pure process based MoC has to be given up. Timing and events have to be integrated into the tripartite modeling approach. Following the tripartite separation principles, this has to be performed in channels. Thus, channels have to be defined, which connect a module to an event. In this way, events can be added while computation can be kept purely untimed.

This extension leads to two channel categories, see Fig. 4.18. On the one hand communication channels are used to actually exchange data between two modules. On the other hand timing channels are used to either generate or receive events. Fig. 4.18(a) shows two basic timing channels for the reception and the generation of a single event. Also communication channels can be extended by the integration of events. Two basic communication channels are shown in Fig. 4.18(b). The first one, marked (1), denotes an untimed point-to-point communication. The second channel, labeled (2), synchronizes the data exchange to an event. Of course, more complex channels are possible. However, the illustrated examples denote four principle channel structures.

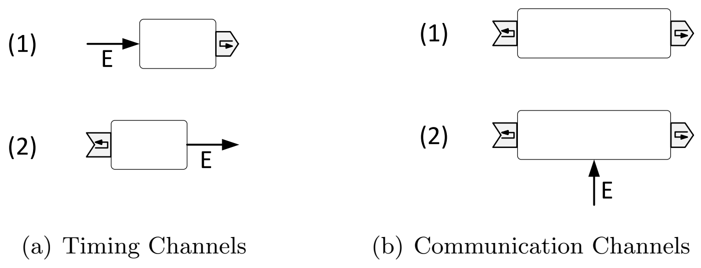


Figure 4.18: Four principle channels structures including events. Timing channels are used to either generate or receive an event. Communication channels are used for untimed or synchronized point-to-point data exchange between connected modules.

Fig. 4.19 shows an example system modeled using the TDA. It consists of four HWSW-Module (M1 to M4). Three of them utilize a CTL container. Since the modules are implemented completely untimed, they have to have at least one blocking input or output. In Fig. 4.19 the blocking inputs are marked using the letter "B". The four modules are connected via four untimed communication channels (C1 to C4). Additionally, two timing channels T1 and T2 are used. T1 is used to synchronize M3 to the input event E1 and T2 generates an event E2. Such events can be any signal used to synchronize different parts of the system.

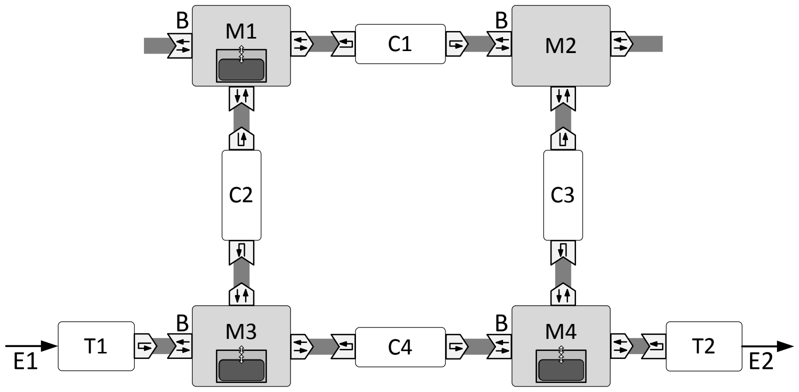


Figure 4.19: Part of a simple example system modeled using the TDA.

This first model of a system is completely independent of any intended implementation. It utilizes solely high level design components and it is used to verify its correct functional behavior. In the subsequent design space exploration different application-to-architecture mappings are explored.

4.3.3 Refinement and Partitioning

To find the best hardware/software partitioning, a design space exploration is performed. Thus, several refined system models with different HW/SW partitionings are generated. According to the partitioning, parts of the functional model are refined to a synthesizable hardware model and parts are refined to a compilable software model.

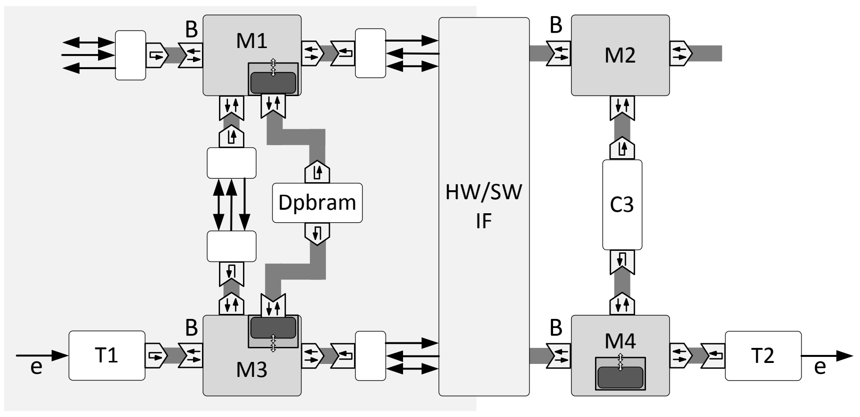


Figure 4.20: Part of a simple system modeled using the TDA.

Fig. 4.20 shows a refined model of the example system, which has been shown in the previous Section. The presented mapping is one of many possible realizations. In this refined model the modules M1 and M3 are mapped to hardware and M2 and M4 are mapped to software. To refine M1 and M3 to hardware modules, data types have to be replaced by synthesizable hardware data types. The bit width is reduced to save hardware resources. Preprocessor macros are utilized to transform the high level HWSW-Module to a synthesizable module. The CTL container implementations are replaced by specific hardware implementations. Furthermore, the containers have to be mapped to actual memory structures. The CTL provides three different memory structures: Ram, Bram and Dpbram. For further details see Sec. 4.2.2.3. In Fig. 4.20 the hardware containers are mapped to the ports of a dual port block RAM. Thereby, one memory structure is efficiently used by two different data structures.

To form a synthesizable module structure, the channels are replaced by adapters, which transform the high level function interface to a low level pin interface. The actual communication among modules can be performed directly by connecting the adapters to each other, see Fig. 4.20. However, different other communication structures can be utilized. It is e.g. possible to connect the adapters via point-to-point communication structures, like FIFOs, or to connect several modules using a bus system. Also timing channels are replaced by low level components. As shown in Fig. 4.20 it is possible to directly use an adapter to connect an event to a module. Another possibility is the use of an explicit hardware unit, which is connected to the module via an adapter. An example for a hardware timing channel receiving an event is a counter, which measures a particular time span. On the other side, a numerical controlled oscillator (NCO) is an example for a hardware timing channel generating an event.

The modules M2 and M4 are mapped to software. Typically, the different computation threads are executed in parallel atop an operating system. Therefore, the module's data types have to be replaced by software data types and preprocessor macros are used to remove SystemC specific keywords to transform the HWSW-Module to a pure C++ class. The implementation of the CTL container used by M4 is replaced by a software specific implementation. Currently, the CTL provides dynamic implementations, which are internally based on STL containers, and static implementations.

As for hardware, communication and timing channels have to be replaced with software specific channels. Comparable to the high level model, a point-to-point communication structure is used. The communication mechanisms are implemented utilizing facilities provided by the underlying operating system. Therefore, communication channels are often operating system specific. Timing channels are replaced by software components, which utilize certain processor components to receive or generate an event. These processor components can be general purpose input/outputs (GPIO) to exchange synchronization signals with other processors or a hardware timer to measure time or to generate a periodic event.

Finally, a communication interface for the HW/SW interface has to be chosen. This can be any serial or parallel communication interface supported by the used processor. Examples are an address/data bus interface or a serial bus system like I2C [54]. On the hardware side, the communication adapters are connected to a low level synthesizable interface implementation. In software, the ports of M2 and M4 are connected to adapters, which translate the function calls to calls to the drivers of the utilized peripheral.

In this way several different HW/SW partitionings can be generated easily. These post partitioning models denote actual HW/SW implementations, which can be directly processed by synthesis tools or compilers.

4.3.4 Design Evaluation

To evaluate the different generated design solutions, various design parameters like cost, performance or power have to be estimated. Design parameter estimates are then utilized to verify the fulfillment of design constraints and to classify the different partitionings in the design space. This classification helps the designer to choose the best partitioning for the given application.

Table 4.4: Different low level tools and design steps are utilized to accurately estimate design parameter. The Table indicates the design parameters, which are provided by the respective tool or design step.

	Costs	Performance	Component Utilization	Power
High Level Synthesis	●	●		
Logic Synthesis	●			
Simulation		●	●	
Prototype		●	●	
Power Estimator				●

In Tab. 4.4 various low level tools and design steps are listed. The Table indicates the type of design parameters, which are generated by the respective design step. Design components mapped to hardware are first synthesized using a high level synthesis tool. The CTL is optimized for the Cynthesizer from ForteDS [10]. However, the library can easily be adapted for other HLS tools. Outputs of an HLS run are estimates of the required hardware resources and the exact latency of each design component. Hence, the HLS tool can be utilized to generate an initial estimate of the costs and performance. The primary output of an HLS tool is RTL code. On the one hand, VHDL or Verilog code is generated for further synthesis. This code can directly be synthesized by a logic synthesis tool, like XST from Xilinx [24], which provides the exact hardware effort. Additionally, many HLS tools generate RTL SystemC code for low level simulation.

The estimated costs for software depend on the performance, which can be achieved with a certain processor. The processor has to be powerful enough to fulfill design constraints. On the other side, a performance overhead is also not desirable, since a more powerful processor is usually more expensive. The measured software performance is therefore directly related to the overall system costs. Software performance can be measured via simulation or by using a prototype. If an instruction set simulator (ISS) is available for the desired processor, it can be utilized to measure the execution time of different parts of the program. The same can be performed by using a prototype via measurement and in-system-debug mechanisms.

Furthermore, both, simulation and prototyping can provide information about the utilization of different processor components. This information can then be used to apply low level power estimators. Often, chip vendors provide power estimation spread sheets, e.g. TI [59], which estimate the power consumption based on the used voltage, the clock frequency and utilization information. Hardware power estimators like XPower from Xilinx [67] use the netlist generated by the logic synthesis tool and signal switching rates obtained via simulation to estimate the power consumption of different components of the systems.

This overview showed that low level hardware and software tools provide accurate estimates of different design parameters. Thereby, HW/SW partitionings generated via the Tripartite Design Approach can be classified and evaluated. This enables an efficient exploration of the HW/SW design space and it helps the designer to find the best application-to-architecture mapping for the given design constraints.

4.4 Summary and Evaluation

An ESL environment, which supports design space exploration, has to enable the generation of different application-to-architecture mappings out of one and the same high level functional model. It has to be possible to map single design components of this high level model to hardware and software without much design effort. In other words, the design components have to be implemented realization independent, so that they can be switched from hardware to software and vice versa. Thereby, a high abstraction level is as important as the efficient mapping to the actual target. Only if the ESL solution generates efficient HW/SW implementations, an accurate evaluation of the mapping decision is ensured.

The presented TDA is based on the tripartite design separation into computation, communication and data structures. In contrast to existing solutions, a high abstraction level is enabled by supporting complex data structures. By separating the design in the mentioned components, computation can be implemented realization independent. In Sec. 4.1 modeling guidelines for the so-called HWSW-Module have been presented. This module encapsulates a single untimed computation thread, which can be mapped to hardware as well as to software without changing the source code. Thereby, an important factor is the utilization of high level synthesis, which significantly reduces the modeling differences between hardware and software for pure computation. The remaining differences are handled by several adaption mechanisms like preprocessor macros and template parameters.

For communication and data structures it is not possible to find a high level implementation, which can be mapped to both an efficient hardware and software implementation. Therefore, a library based approach is applied for these design components. The presented CTL provides commonly used complex data structures and various different implementations, which are optimized for high level simulation, hardware and software design. A mechanism based on static polymorphism allows the simple exchange of the used implementation. Thereby, different implementations can be utilized for several design phases and application-to-architecture mappings. While existing solutions often generate generic implementations, this approach allows target specific optimizations. The currently provided CTL hardware implementations are optimized for FPGAs. Since the efficient mapping to available resources is especially important for FPGA designs, a memory mapping facility has been added. This facility allows the mapping of different data structures to one and the same memory structure.

Following the design principles of SystemC, communication is modeled using channels. The TDA does not only support simple communication channels. It also supports channels, which enable the integration of events. This results in two channel categories: timing and communication channels and it allows the modeling of timing and synchronization functions within the tripartite modeling approach. As a result, the TDA is not limited to pure data flow design. Basically all designs, which can be modeled by using the discrete event MoC, can be implemented via the TDA. Although, this design concept moves a certain amount of complexity into channel design, the modeling freedom denotes an advantage compared to many existing design solutions, which are limited to a specific application domain.

With the TDA modeling and refinement concept various application-to-architecture mappings can be generated easily. Then, low level tools, simulation and prototyping can be applied to estimate critical design parameters such as power, performance and costs. In this way, several solutions to a system level design decision, like the application-to-architecture mapping, can be evaluated and compared. The designer can explore the design space and find the best solution with a relatively low effort.

5 Case Study: HW/SW Codesign of a VoIP Engine

In the previous Chapter, the Tripartite Design Approach has been presented. By using this design principle along with the developed Codesign Template Library, the realization independent design of high level computation components is possible. This simplifies hardware/software codesign and design space exploration. To proof its applicability and to evaluate the reduced design effort the design approach is in the following applied to a real world example.

As case study the realization of an embedded VoIP engine has been chosen. One reason for this decision is the VirtualVoIP research project financed by Frequentis AG [12]. The proposed design flow is one outcome of this project. Beside these more practical reasons, it is an interesting example since it does not only consist of pure data flow components. The presented sample rate converter is a solution to a synchronization problem as they appear in real world examples. It requires timing and synchronization components, which cannot be modeled using pure data flow approaches, e.g. [HSKM08]. This shows that the TDA can be applied as well to such heterogeneous designs, which is a strength compared existing solutions.

In the following, the design goals and principle functionality of the VoIP engine are illustrated, see Sec. 5.1. One part of it, the Real-time Transport Protocol (RTP) engine, is then designed using the Tripartite Design Approach. Details thereto are shown in Sec. 5.2. The functionality of this part is verified in Sec. 5.3 by several tests and measurements. Then, in Sec. 5.4, different hardware/software solutions are compared concerning their power consumption and their costs. The design effort is evaluated and compared to a traditional design approach in Sec. 5.5. Finally, the case study is summarized and the results are briefly evaluated and discussed in Sec. 5.6.

5.1 An Embedded VoIP Engine

First attempts to transmit voice digitally over a network reach back to the beginnings of the Internet. In August 1974 a successful transmission of digitalized voice over the predecessor of the Internet, the Advanced Research Projects Agency Network (ARPANET), has been performed. Back then, it already became apparent that voice transmission requires different conditions than data transmission. This fact significantly influenced the transmission protocol development of the Internet. Originally planned as one protocol, the Transmission Control Protocol (TCP) [RFC81b] and the Internet Protocol (IP) [RFC81a] have been separated, whereby IP acts as the basis for all Internet services. Specifically for voice transmission, the development of the simpler User Datagram Protocol (UDP) [RFC80] started [Gra05] in 1977.

However, it still took several decades until the Internet protocol has been used for realtime voice transmissions on a larger scale. One reason for the delay were the costs of the limited bandwidth which cheapened significantly not until the last decade. Another reason were the problems that emerge from the switch from a circuit switched to a packet switched network for realtime transmission. Only the development of specific protocols, which handle these problems enabled the acceptance of VoIP. Eventually, the technology gained popularity by the success of the free VoIP provider Skype [29] in the last decade. Most recently, Internet telephony has become a cheap alternative to public switched telephone networks.

The lower acquisition and operational costs and the possibility to provide not only voice but also different other services over one and the same network accelerates the replacement of circuit-switched by packet-switched networks. Recently, VoIP systems are increasingly installed also in safety critical application areas like emergency call centers or air traffic management systems. However, this specific application areas demand particular requirements. Whereby, it is not possible to utilize standard solutions like they are used for example in the consumer market.

In this case study a stereo VoIP engine was developed, which is optimized for the design constraints given by safety critical applications. In the following sections the functional requirement of an embedded VoIP engine are derivated, whilst taking into account the specific requirements of safety critical applications.

5.1.1 Design Goals and Constraints

In this Section, the design goals and constraints of this case study are defined. These objectives consist on the one hand of goals which are important for every system design today and on the other hand are derived from the specific requirements of safety critical applications.

Beside availability, reliability, and system redundancy, safety critical application areas demand also specific requirements to the voice quality [32]. An important factor for instance is the end-to-end delay [31], which is defined as the delay from the point in time when the speaker's voice is recorded until the voice signal is actually played at the intended listener. Especially for safety critical applications it is important that the communication is not disturbed by too long transmission delays. The ITU-T recommends a maximum end-to-end delay of 150 ms [Int03] for high quality voice transmissions. This requirement influences not only the development of the network infrastructure but also the end device development. Therefore, the latency produced by the processing in the end device should be minimized.

Compared to consumer electronics, niche markets like air traffic management or emergency call centers are small markets. Communication systems for such markets are expected to be sold only in small to medium quantities. This influences the design of such systems. It is not profitable to design a complete SoC nor is it necessary since the advantage of SoCs, the size, is not a significant design criteria for such systems. Rather components off-the-shelf (COTS) like FPGAs and general or special purpose processors should be used.

For mobile devices the power consumption is extremely important. However, power efficient system design is an issue for all systems today. Due to the raising energy consumption costs, it is possible that the system's power consumption heavily influences the purchase decision. Therefore, the minimization of power should be a design goal for the present case study.

Last but not least, the costs should be minimized. In fact, for each design decision both production and development costs have to be consider. However, production as well as development costs are difficult to estimate in this case study. Therefore, component costs should be minimized. Accordingly, the following design constraints for the embedded VoIP engine can be outlined:

- Minimum component costs,
- Minimum power,
- Minimum end-to-end delay,
- Use COTS.

5.1.2 Voice Transmission over the Internet Protocol

Beside proprietary VoIP systems like Skype publicly accessible protocols are available. These protocols define the general process of VoIP calls and ensure the compatibility of different systems. Different protocol classes can be distinguished. Each protocol class consists of several quasi competing protocols which define a certain functionality required for a VoIP connection.

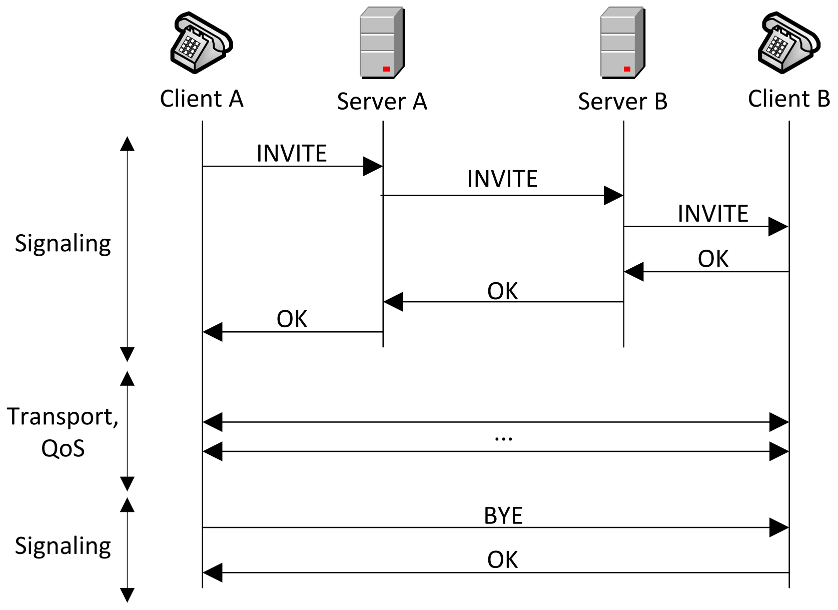


Figure 5.1: Basic communication sequence of a VoIP call between Client A and Client B.

In the following, protocol classes which are required for VoIP are illustrated using a simple VoIP session, cf. Fig. 5.1. Client A wants to call client B. Typically, the client first contacts the server where it is registered (server A). This server knows that client B is registered at server B, which in turn knows the exact location, hence the IP address, of client B. Via this route an initially INVITE message is sent to client B. At that time the phone of client B starts ringing. After the callee picks up the phone, an OK message is sent back the same route. This part of the call, the call set-up is defined in the signaling protocols. The presented sequence is only an example. Also an immediate direct communication between A and B is possible if the caller knows the IP address of the callee a priori.

After call set-up audio data is exchanged directly between the two communication partners in a digital way. This part of the call is defined by the transport protocols. During the exchange of voice packets so-called quality of service information is exchanged. Thereby, the receiver gives the sender feedback about the quality of the received data. This is necessary since the transmission quality may change over time in packet switched networks. Finally, the participant who wants to end the conversation hangs up the phone. Then, the corresponding client sends a BYE message, which is acknowledged by an OK. The part of ending the call is again defined by the signaling protocols.

5.1.2.1 Signalling Protocols

As already mentioned, signaling protocols define the set-up and the ending of a VoIP connection. It is possible that the caller initially only knows a special VoIP alias. However, for a peer-to-peer connection the actual IP address is necessary. As shown in Fig. 5.1 the IP address can be obtained by using appropriate servers. Additionally, it is required to exchange information about the following peer-to-peer communication. For example it is necessary to negotiate the sampling frequency with which the audio data is digitized or the used voice codec, to compress the voice packets.

According to [FCP09] there are two types of signaling protocols: peer-to-peer and master-slave protocols. In this work only the more common peer-to-peer protocols are discussed. Currently, three major protocols exist: H.323, SIP and Jingle. The oldest protocol is H.323 which has been released by the ITU-T in 1996 [Int09]. It is not just a single protocol rather a set of protocols. The call set-up is defined in the Q.931 standard, which is a variation of the Q.931 used in the ISDN [Int80]. Thus it is well suited for interworking scenarios between IP and ISDN networks [LM00].

The second signaling protocol, the Session Initiation Protocol (SIP), has been proposed by Schultzrinne in 1999 [RFC02]. Today, it is one of the most used VoIP signaling protocols. Its general structure is simpler than the structure of H.323. The protocol is optimized for communication demands of modern IP networks like dynamically changing IP addresses. The last and most recently defined protocol is Jingle [SA07], which has been introduced in 2004. It is basically an extension to the extensible messaging and presence protocol (XMPP), which is mainly used for instant messaging and became famous with the Jabber [27] messenger.

5.1.2.2 Media Transport Protocols

In the Open Systems Interconnection (OSI) reference model [ISO94] transport protocols denote protocols at the transport layer like TCP and UDP, which are defined atop IP. Media transport protocols are one layer above and provide specifically the transport of media data like voice. At the transport layer mainly UDP is deployed, since according to [AGSS09] it currently is the best suited protocol for media transportation. Besides proprietary solutions, the only media transport protocol in use is the Real-time Transport Protocol (RTP) [RFC03b]. All mentioned signaling protocols specify the use of RTP as media transport protocol [JPPG04, p. 229-251], [SA05, SR00].

The RTP protocol defines sequence numbers and time stamps. The sequence numbers can be used to restore the packet order in case they are not received in proper sequence. Using the time stamps it is possible to estimate the current network jitter. Additionally, the RTP protocol possesses several profiles, which define the transport of different media data encoded with different codecs.

5.1.2.3 Quality of Service

The third class of protocols concern the quality of service (QoS) [ZOS00]. In contrast to circuit switched networks parameters of the communication channel may change over time in packet switched networks. Due to the separate routing of each packet it is possible that the transmission time changes for each packet. This variation is also called network jitter. Furthermore, the bandwidth for a communication link is not guaranteed, it rather depends on the current network load. To account for these issues it is required to constantly observe the connection and its quality. The Real-Time Transport Control Protocol (RTCP) [RFC03a] is a quality of service exchange protocol which is associated with the RTP protocol. It defines so-called reports which are sent constantly by each communication partner. Via these reports information like the number of lost packets or the estimated network jitter are exchanged. RTCP defines only the information exchange but it does not define functions to improve these connection parameters.

To actually ensure and improve QoS parameters several techniques like the prioritization of MAC or IP packets [ZOS00] have been developed. However, these techniques are not described in more detail in this work.

5.1.3 Arbitrary Sampling Rate Conversion in VoIP Systems

A problem in VoIP applications are the slightly different sampling rates at the sender and receiver side. Although the sampling rates of the analog-to-digital converter (ADC) at the sender and of the digital-to-analog converter (DAC) at the receiver are initially negotiated, the used sampling rates are never exactly the same. Since the sender and the receiver are different systems at different locations, the desired sampling rate is generated from different clock sources. This leads to nominally equal but slightly different sampling rates.

The transmission of samples over an IP-based network introduces a network jitter. To remove this jitter a buffer is necessary at the receiver. If the sampling rate of the receiver is slightly slower than the sender's rate a buffer overflow will eventually occur. If the sender's clock is slower, then a buffer underrun will occur at the receiver. In the first case, it is necessary to flush the whole buffer, which means that a part of the received audio signal is discarded. If an underrun occurs, silence is inserted. In both cases a disturbance will occur, which is not acceptable for safety critical communication systems used for air traffic management or in public transport systems.

PC-based VoIP systems reduce this problem by enlarging the buffer. This reduces the occurrence of the buffer overflows and underruns, but at the same time enlarges the latency of the whole system. This is also not acceptable for safety critical systems. Thus, the buffer overflow and underrun problem is solved by recovering the sender's clock at the receiver site. Then, the jitter buffer can be read out with the sender's clock. In this way, a buffer overflow or underrun is avoided and the buffer can be kept very small. Anyway, it is necessary to convert the samples from the sender's clock domain to the receiver's domain. If no conversion is performed either at least one sample is taken twice or at least one sample is lost. With a typical frequency stability of 100 ppm [9] and a sampling rate of e.g. 8 kHz, every 1.25 seconds a short time broadband noise is introduced in the output signal. Of course, it is possible to use high accuracy oscillators with a better frequency stability, but this only reduces the problem. It does not solve it.

The operation of converting the signal between the clock domains is basically a resampling process. Since the sampling rates are almost equal and since the conversion ratio may change over time or from connection to connection, common resamplers cannot be used. It is necessary to use special resamplers for arbitrary sampling rates, which are able to handle changes of the conversion ratio during runtime [BWH09].

5.1.3.1 Conversion between Arbitrary Sampling Frequencies

Resampling a signal to a new sampling frequency denotes the process of calculating new sample values in between the original samples. If the relation between the sampling rates is a rational factor, resampling is usually performed by interpolating the signal by an integer factor, filtering the signal to avoid aliasing and finally decimating the signal by another integer factor [CR83]. To use this method, the ratio has to be known in advance.

In this work, the resampler has to convert between two almost equal sampling frequencies with a ratio being maybe different for every VoIP call. Therefore, it is necessary that the system is self adapting to every new frequency ratio. There are several different methods presented in [Ram84] to convert a signal between arbitrary sampling frequencies. A simple method has first been presented in [LPW82]. Lagadec *et al.* proposed to interpolate the signal to a very high frequency. The output signal is then generated by taking the closest sample to the correct sampling instant. The higher the sampling frequency F_s, to which the signal is interpolated, the smaller is the error in the output signal. Ramstad has shown in [Ram84] that the error in the output signal is smaller than the quantization error if the inequation

$$F_s \geq \pi \cdot 2^{b+1} F_M \tag{5.1}$$

is fulfilled, F_M denoting the highest frequency of the signal and b denoting the bit width. Many VoIP systems use pulse code modulation (PCM) like G.711 [Int90] with a sampling rate of 8 kHz. These systems encode 13 bit samples to 8 bit with a logarithmic characteristic. It is assumed in this work that the voice signal is band limited with a maximum frequency of 4 kHz. According to Eqn. 5.1 this band limitation leads to an interpolation frequency of about 200 MHz for a 13 bit system.

If linear interpolation between the neighboring samples is included, as presented in [LK81], the inequation changes to

$$F_s \geq \pi \cdot 2^{(b+1)/2} F_M \tag{5.2}$$

according to [Ram84]. This reduces the needed interpolation frequency to about 1.6 MHz, but the computation effort for the interpolation filter would still be tremendous.

A different approach is presented by Smith in [SG84]. An arbitrary digital signal $x(nT_s)$ with a sampling frequency $F_s = 1/T_s$, n ranges over the integers, is assumed to be bandlimited to one half of the sampling frequency. Due to Shannon's sampling theorem it is possible to reconstruct the original signal using

$$x(t) = \sum_{n=-\infty}^{\infty} x(nT_s) h_s(t - nT_s), \tag{5.3}$$

where

$$h_s(t) = sinc(F_s t) = \frac{sin(\pi F_s t)}{\pi F_s t}. \tag{5.4}$$

Eq. 5.3 basically denotes a convolution of the digital signal with a continuous Sinc function. The Fourier transform of the Sinc function is a rectangle. Thus, the convolution of $x(t)$ and $h_s(t)$ corresponds with a filtering process with an ideal low pass filter $H_s(f)$ in the frequency domain. Thereby, the image spectra of the periodic spectrum $X(f)$ are removed and the original spectrum of the continuous signal is reconstructed, as shown in Fig. 5.2.

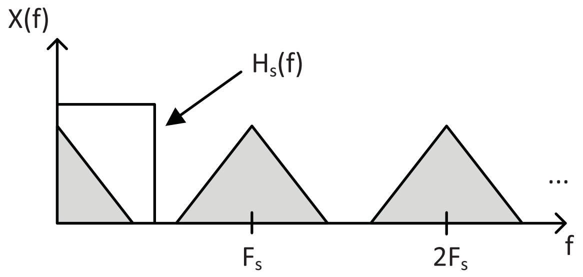


Figure 5.2: $X(f)$ is the periodic spectrum of an arbitrary band limited digital signal with a sampling frequency F_s. The ideal reconstruction filter $H_s(f)$ removes the image spectra of $X(f)$ to reconstruct the original spectrum of the continuous signal.

If this signal has to be resampled to the sampling frequency $F'_s = 1/T'_s$, then Eq. 5.3 only has to be evaluated at the sampling instants of the new sampling frequency

$$x(mT'_s) = \sum_{n=-\infty}^{\infty} x(nT_s) h_s(mT'_s - nT_s). \tag{5.5}$$

5.1.3.2 Filter Design

Using Eq. 5.5 the original voice signal can be resampled ideally. Obviously, this is not realizable, as this would require to solve an infinite sum. A continuous impulse response h_s would be required as well, since the actual filter coefficients are determined by the permanently changing phase difference of the two clock domains, see $h_s(mT'_s - nT_s)$ in Eq. 5.5. A realizable approximation can be achieved, if a digital filter with a finite impulse response (FIR) is used as reconstruction filter. Several methods exist to design digital FIR filters [OSB99]. For the resampling purpose it is necessary to significantly oversample the filter to get an almost continuous impulse response. According to [57] the window method using the Kaiser window is a very simple and robust method ideal for high sampling frequencies [KS80].

To find the ideal sampling frequency for the filter different design trade-offs have to be kept in mind. On the one hand, the sampling frequency has to be high enough so that the error due to the discrete impulse response is smaller or equal to the quantization error. On the other hand, higher sampling frequencies demand a larger memory to store all the filter coefficients. The accuracy of the filter coefficients can be increased if the values actually used are calculated during runtime by a linear interpolation of two consecutive, stored coefficients. This significantly reduces the needed sampling frequency.

In principle, the sampling frequency of the impulse response can be interpreted as the sampling frequency, to which the signal is first interpolated before it is sampled with the new sampling rate.

Hence, an interpolation is followed by a decimation. The difference is that interpolation values, that are removed by the decimation, are not calculated in first place. Therefore, Eq. 5.2 can be used to estimate the appropriate frequency. To simplify the implementation, the interpolation frequency also should be an integer multiple of the sampling frequency of the audio data.

The VoIP system, which is designed in this case study uses PCM encoded data using the A-Law codec and a sampling frequency of 8 kHz. Detailed information and the general structure of the system is shown in 5.1.4. Based on the PCM codec, the samples have an accuracy comparable to 13 bit linear data. Using Eq. 5.2 leads to a minimum sampling frequency of 1.6 MHz for the filter. This calculation is heavily pessimistic. For simplification the sampling frequency is set to 1 MHz.

Another design decision is the length of the impulse response. With a longer impulse response a steeper filter with a larger stop band attenuation can be designed. However, a longer impulse response increases the overall latency of the filter. Since this work focuses on safety critical communication systems, the latency should be as small as possible.

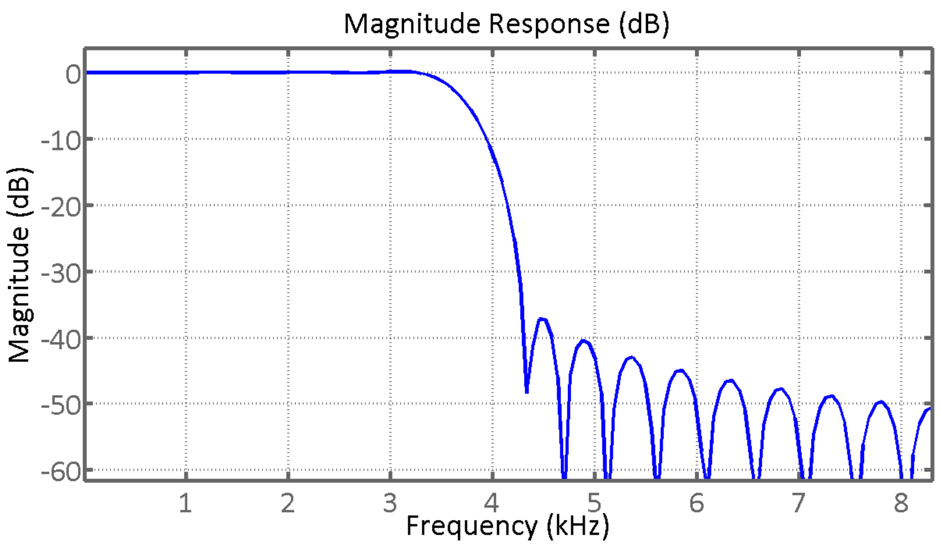


Figure 5.3: Magnitude response of the FIR filter function used in the resampler. The filter has been designed for an audio signal with 8 kHz sampling frequency. The utilization of aliasing filters is presumed, which allows a wide transition band from 3.4 kHz to 4.6 kHz.

The implemented impulse response is a filter with an order of 2002. With a sampling frequency of 1 MHz the filter has a latency of approximately 1 millisecond. The filter has been designed with the window method using the Kaiser window. Due to the ITU-T recommendation G.712 [Int02] the transition band starts at 3.4 kHz. Since it is assumed that aliasing filters already attenuate the frequency band between 3.4 and 4 kHz, the first mirror image starts at 4.6 kHz. Thus, the transition band can be enlarged to 4.6 kHz. The cut-off frequency of the filter is set to 3.8 kHz and the beta value of the Kaiser method is 3. This value influences the style of the Kaiser window. A larger beta value enlarges the transition band of the filter, but it also reduces the ripple in the stop band. The magnitude response of the filter is shown in Fig. 5.3. Pass band ripples are smaller than 1 dB. At 4.6 kHz the attenuation is around 40 dB.

5.1.3.3 Structure of the Resampler

Eq. 5.5, which describes the process of sampling rate conversion for arbitrary sampling rates, has been derived in Sec. 5.1.3.1. Using a finite impulse response, this equation denotes an adaptive FIR filter. Following, an impulse response for this filter with a length of about 2 ms has been designed and presented in Sec. 5.1.3.2. With a sampling rate of 8 kHz, hence a period of 125 μs this results in a filter with 16 taps. The actual coefficients depend on the current phase difference of the two clock signals.

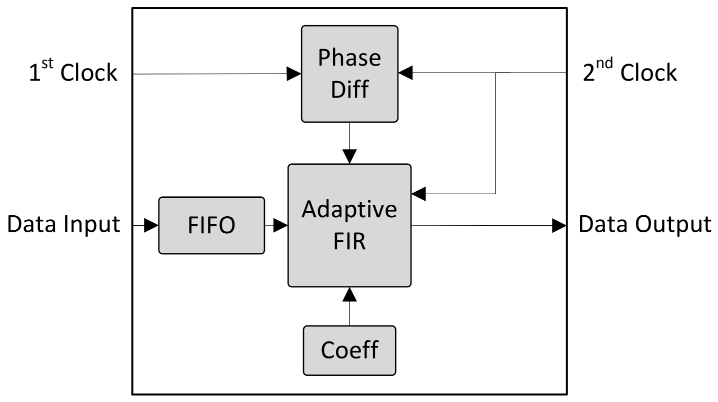


Figure 5.4: The phase difference between the clock domains is measured in the "Phase Diff" unit and is used by the adaptive FIR filter to load the corresponding coefficients ("Coeff"). At the input a FIFO is used to buffer the input data. On every clock edge of the 2^{nd} clock a new output value is calculated.

This leads to a resampler with the structure shown in Fig. 5.4. It consists of a data input and a data output. The input provides audio samples synchronous to the input clock. These samples are stored in a FIFO since it is possible that several input samples are written before a new output value is calculated. At each cycle of the output clock, the adaptive filter reads all available samples from the FIFO and calculates a new output value. The FIFO may contain more than one sample if the frequency of the input clock frequency is higher than the output clock frequency.

For each output value calculation the filter needs the current phase difference of the two clock signals, confer $h_s(mT_s' - nT_s)$ in Eq. 5.5. Thus, a separate unit is required which measures the phase difference at each output clock cycle. Depending on the phase difference, the filter reads the appropriate coefficients from a memory which stores the impulse response $h_s(t)$. Since the impulse response is a symmetric function, it is sufficient to store only one half of it, which results in 1002 coefficients.

5.1.4 Basic Structure of the VoIP Engine

The majority of the functional requirements for the stereo VoIP engine are given by the chosen network protocols. Due to their simplicity and popularity, the protocols SIP and RTP/RTCP have been selected. Additional requirements are based on the given design constraints. The resultant basic functional structure of the VoIP engine is shown in Fig. 5.5. In the following, all components and their interactions are presented in more detail.

The VoIP engine has three different interfaces. First, it is connected to the Ethernet [oEI80], which is used as physical layer in this case study. Typically, a physical layer chip (PHY) handles the analog interface. This chip provides a digital interface, which is connected to a component, that implements the medium access control (MAC) layer. Second, the interface to the user, which configures the VoIP engine and decides to make a call, end a call or to answer a call. Third, the audio interface, where the voice is recorded and converted to digital audio samples. On the receiving path, the audio samples are converted back to an analog signal and are sent to speakers. All digital-to-analog conversion and analog-to-digital conversion is usually performed in an audio codec chip. As sampling frequency for the audio interface, 8 kHz has been chosen. This is a typical sampling rate for telephone applications, as it is used e.g. in GSM [ETS90] and ISDN [Int80] systems. The codec chip provides a digital interface to the rest of the VoIP engine. The counterpart implementing this interface is the "Codec Interface" unit. The illustration in Fig. 5.5 is confined to the pure digital components. Hence, the PHY and the audio codec chip are not displayed.

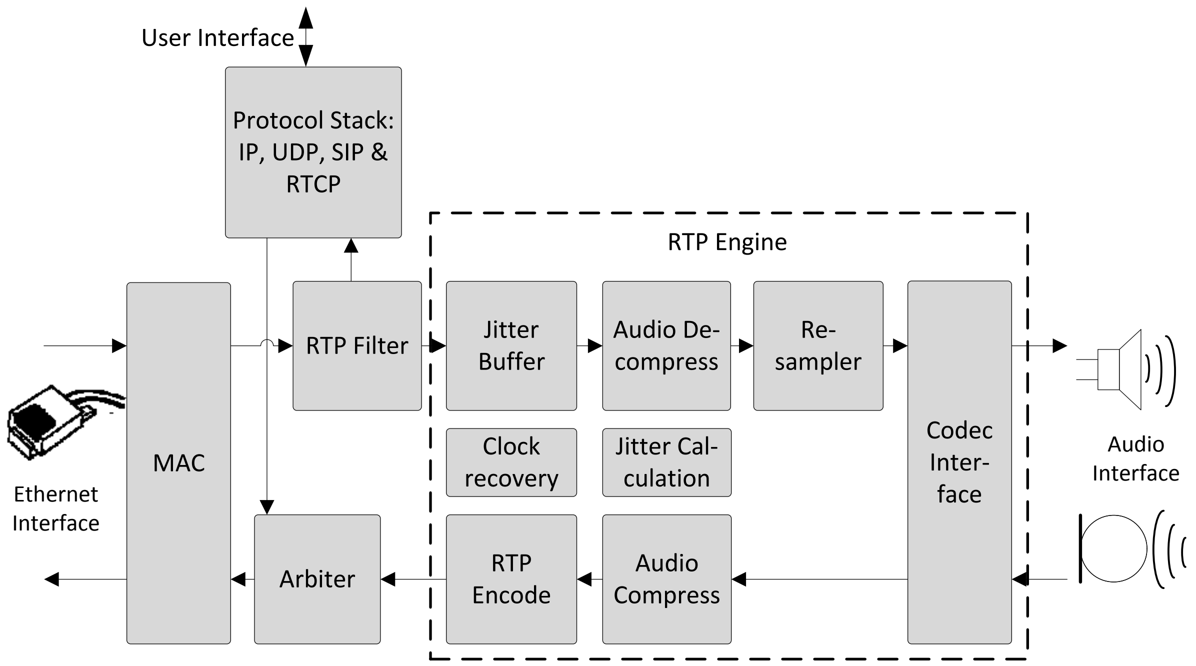


Figure 5.5: Functional structure of the VoIP engine. Signaling and QoS protocols are implemented in the "Protocol Stack". The components, which realize audio transport related functions are grouped as RTP engine.

The system can be divided into two parts: the control flow dominated part and the data flow dominated part. The control flow dominated part is basically composed of the protocol stack of IP, SIP and RTCP. It is shown in Fig. 5.5 as a single block. The data flow dominated part corresponds to the processing chain from receiving or sending an RTP packet to audio recording and playback, see "RTP Engine" in Fig. 5.5. This part is called RTP engine. At the beginning of the RTP engine an RTP filter is required, which separates the RTP packets from all other received packets. During the filter process several protocol header fields are checked, like source and destination of IP and UDP as well as the synchronization source identifier (SSRC) of the RTP header. The remaining packets are solely RTP packets from the current session. Furthermore, the filter removes all header and checksum fields from these packets. Only the audio samples are forwarded to the RTP engine. The counterpart to the filter at the sender side is the arbiter, which controls the access to the MAC interface for sending Ethernet packets. If the RTP engine and the Stack implementation want to send a packet at the same time, the access is granted to the RTP engine. The reason therefore is, that the RTP Engine has to fulfill harder realtime requirements than the Stack implementation.

The RTP engine itself can be divided into the sender and the receiver part. Whereby, beside the codec interface, the sender part of the RTP engine consists of the audio compression and the "RTP encode" unit. For compression, the G.711 A-Law [Int90] codec is used. This is a pulse code modulation with a logarithmic characteristic. It maps 13 bit linear audio samples to 8 bit compressed samples. The advantage of this simple codec is that only one sample is required for compression. Many other codecs with better compression rates require several audio samples for each compression process, which increases the overall end-to-end delay. The same codec is used e.g. in ISDN systems.

The RTP encode unit collects audio samples until the next RTP packet can be sent. The majority of the RTP header is provided by the stack implementation. Only some RTP fields have to be set by the RTP encode unit. The UDP checksum has to be recomputed for each packet, since it is calculated using header and data fields of the packet. The design of the RTP encode unit influences the overall end-to-end delay as well. The default number of samples per packet for G.711 with 8 kHz sampling rate is 160 [RFC03c]. This would result in a 20 ms delay. Eventually, the number of samples per packet has been reduced to 10, which results in a delay of 1,25 ms. Obviously, this increases the network traffic. However, it significantly reduces the end-to-end delay.

At the receiver side, the audio samples, which have been unpacked in the filter, are then written to the jitter buffer, which is necessary to compensate the network jitter introduced by the packet switched IP network. The buffer only can hold up to 32 audio samples per channel, which corresponds to 4 ms. This is a very small buffer size compared to PC based VoIP solutions, e.g. Skype uses more than 200 ms [WCHL09]. However, the reason for the decision was again the minimization of the end-to-end delay. Samples are read out of the buffer, when it is at least half filled. This means that two packets have to be received until the readout starts. Thus, the jitter buffer adds a delay of 1,25 ms to the overall system.

As mentioned in Section 5.1.3, the received audio samples are read out of the buffer synchronous to the senders clock frequency to avoid buffer overflows and underruns. Therefore, the sender's clock has to be reconstructed at the receiver. This can be done by continuously reading the buffer level at a certain point in time, e.g. every time a packet is received. These buffer level measurements are then averaged to filter out the network jitter. The tendency of the buffer level, then represents the frequency difference between the sampling rate at the sender and the reconstructed clock at the receiver. If for example the buffer level increases, the frequency of the reconstructed clock has to be increased slightly. An optimal clock recovery only can be found via a control unit design process. For the purpose of a first prototype and for design space exploration a simple design is sufficient.

The audio samples, which are read out of the jitter buffer synchronous to the recovered clock, are then converted to linear audio samples in the "Audio Decompress" unit, see Fig. 5.5. Although the utilization of a clock recovery avoids a buffer overflow or underrun, there are still two different clock domains: the sender's domain and the receiver's domain. The conversion of the audio samples between these domains is performed in the resampler, which has been presented in more detail in Sec. 5.1.3.

The final component is the jitter unit. It calculates an estimate of the statistical variance of the packet inter arrival time as defined in the RTP standard [RFC03b]. Therefore, time stamps of the received packets and a local sample counter are used. The calculated jitter is required for RTCP packets.

5.2 System Level Design

In this Section, a system concept of the VoIP engine is developed. First, this requires the definition of the basic architecture of the system. Then the different functional components are mapped to the different ICs. Typically, a system consists of parts, which are clearly control flow dominated and which are therefore ideal for a realization on a GPP. The data flow dominated part may be mapped to SPPs or to hardware. Specifically for these blocks, a high level model using the TDA is implemented. As already mentioned, both a hardware realization and a software realization of such components has advantages and disadvantages and it is difficult to find the partitioning, which fits best to the given design constraints. Except for a pure HW solution, three different HW/SW partitionings are elaborated in the following. Additionally, the requirements for the HW/SW interface are analyzed and its realizability is examined.

5.2.1 Architecture Design

As mention in Sec. 5.1.1 COTs products should be used for the VoIP engine design, since only small to medium quantities are produced. In this case, the architecture design step corresponds to the mapping of the functional components to different ICs like GPP, SPP and FPGA. The control flow dominated part of the RTP engine, the protocol stack implementation, is well suited for a realization in software using a GPP. More difficult is the mapping of the RTP engine, which on the one hand can be mapped to hardware, hence to an FPGA. On the other hand it can be realized on a DSP as software.

In [Wen11] the realization of a VoIP stack on a PowerPC [26] is presented. The target technology in this case is a Xilinx Virtex4 FPGA [65], which has an embedded PowerPC and a media access control (MAC) core hard macro. This enables the resource efficient realization of the VoIP stack on the PowerPC. At the same time, the arbiter and the filter can be easily connected to the processor core as custom hardware peripherals. The configurable nature of FPGAs makes it easy to connect the RTP engine to the processor and the Ethernet interface can be realized using the on-chip MAC core.

The RTP engine could in this case be realized on the same FPGA in hardware or on a connected DSP in software. Both variants have of course advantages and disadvantages. A single chip solution on one FPGA would be smaller and would probably consume less power. However, FPGAs are more expensive compared to DSPs. The overall costs may be reduced if shifting parts of the RTP engine to a DSP enables to switch to a smaller and cheaper FPGA. Whereby, it is difficult to estimate how powerful the DSP has to be to perform the processing with the required data rates. Since the estimation of the power consumption and the required FPGA resources is difficult as well, a design space exploration of this problem is performed.

Xilinx FPGA and Texas Instruments (TI) DSP prototyping boards are already available at the university. They can be used for the design space exploration. Therefore, the architecture is fixed to a Xilinx FPGA and probably a TI DSP. For the FPGA the family Virtex4 is fixed. It is the oldest chip, which has a MAC core and a PowerPC. All newer FPGAs, like the Virtex5 family, are much more expensive. That implies the mapping shown in Fig. 5.6. The protocol stack, the Ethernet interface, the arbiter and the filter are realized on the Virtex4 FPGA. The RTP engine is either realized on the FPGA or partly implemented on a connected TI DSP.

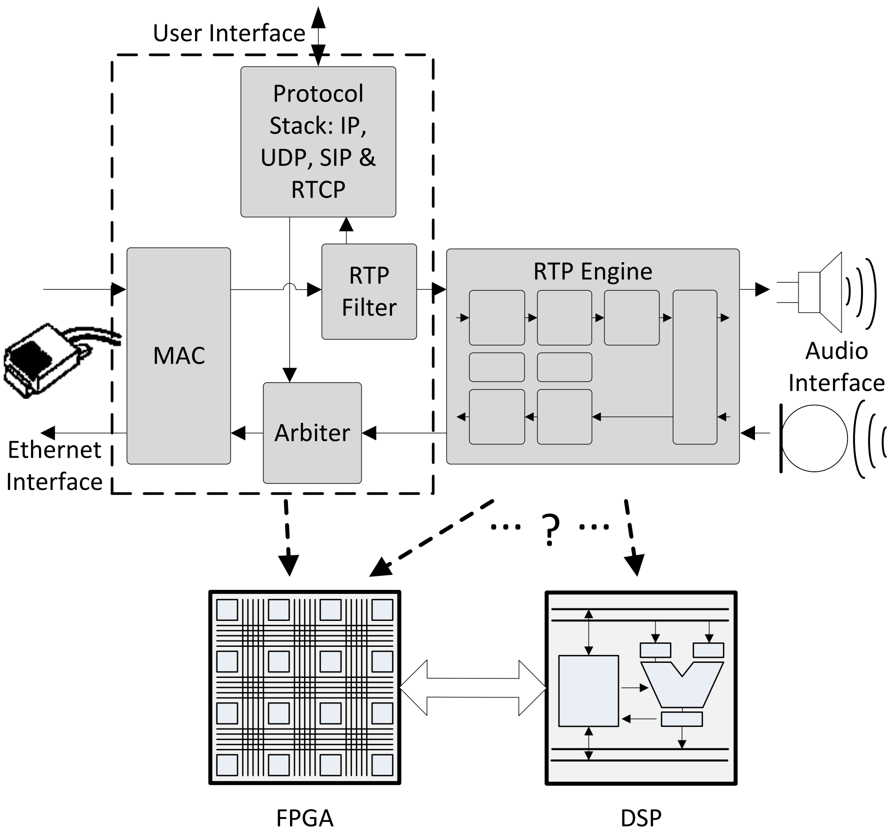


Figure 5.6: Architecture of the VoIP engine consisting of an FPGA and an optional DSP. Ethernet interface, protocol stack, arbiter and RTP filter are realized on the FPGA. The RTP engine is either implemented on the FPGA or on the DSP or it is partitioned and partly realized on both devices.

5.2.2 Tripartite System Design of the RTP Engine

The RTP engine and the stack implementation can be developed separated from each other. The interface between them is composed by the arbiter and the filter. Additionally, the RTP engine is configured by the stack implementation and in turn it provides information like the current jitter. In the following design space exploration, primarily the partitioning decision of the RTP engine should be clarified. Therefore, the VoIP engine is for now designed without the protocol stack. To get a working prototype of the RTP engine, all components from the Ethernet interface to the audio interface except the arbiter are required. The filter is necessary, since it unpacks the RTP packets and it provides the received timestamps, which are used in the calculation of the jitter estimation. The configuration of the RTP engine can be performed statically.

The embedded MAC core of the FPGA can be utilized via a VHDL wrapper supplied by Xilinx. This core handles the media access control layer of Ethernet and it provides a simple so-called Local Link interface to the RTP engine. All other components are implemented in SystemC to get a first system level model. Since the RTP filter is definitely realized on the FPGA, it can be directly implemented in synthesizable SystemC on RTL. The remaining modules are mapped either to HW or to SW and are consequently wherever possible modeled realization independent using HWSW-Modules. Thus, it is possible to test different HW/SW partitionings. Like presented in Sec. 4.1, the HWSW-Module thereby only contains pure computation. All communication and synchronization is realized in channels and for the data structures, CTL containers are used where possible.

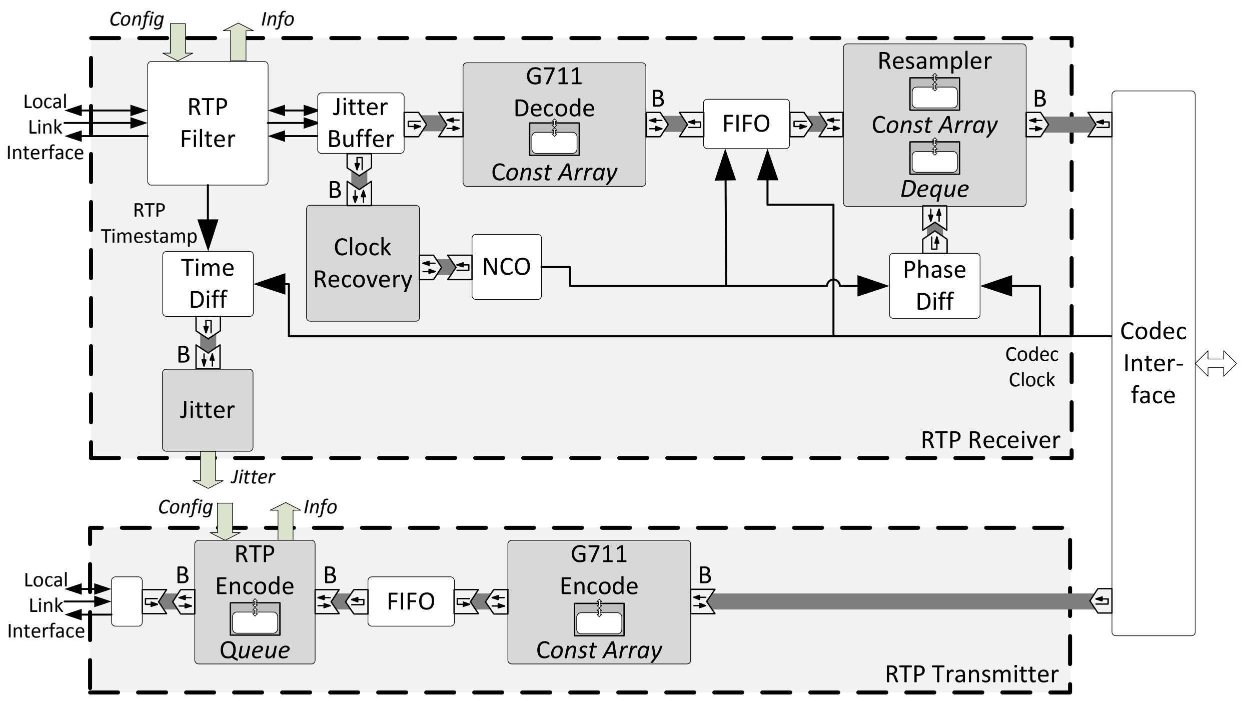


Figure 5.7: System level design of RTP engine. Separated into receiver and transmitter, the RTP engine is modeled using the TDA.

Fig. 5.7 shows the system level design of the RTP engine including the RTP filter. It is separated into the receiver, the transmitter and the codec interface. The receiver and the transmitter posses a Local Link interface, which is connected to the embedded MAC core in the FPGA. Configuration parameters are required for the filter and for the RTP encode unit. Both modules and the jitter unit provide information for the stack implementation. In this prototype design this information is calculated, but is not further processed. The whole system has six HWSW-Modules. The other two modules of the initial RTP engine design have been identified as communication components. The jitter buffer corresponds to a communication channel between the filter and the G711 decoder and the codec interface is the communication channel which connects the RTP engine with the audio codec chip. The codec interface forwards received audio samples to the codec and it provides samples from the codec to the transmitter. Additionally, a clock signal is required, which corresponds to the sampling rate of the ADC and DAC in the audio codec chip. In Fig. 5.7 this clock signal is named ”Codec Clock” and is shown as a signal provided by the codec interface.

The transmitter is significantly simpler. It only consists of two HWSW-Modules. The work cycle is provided by the audio codec. Therefore, both modules have a blocking input. As soon as new audio samples are available, the G711 Encode module encodes the samples using a table, which is stored in a Const Array. The encoded samples are then written to a FIFO, which connects the G711 encode module with the RTP encode module. The RTP encode module collects all samples in a Queue until there are enough samples for a new RTP packet. Since in the final VoIP engine, the Local Link interface can be blocked by the Stack Implementation, it is necessary to store the samples at the input of the RTP encode module in a FIFO. However, the blocking time has to be very short, since the blocking increases the network jitter, which decreases the quality of the VoIP connection. Therefore, the FIFO can be dimensioned very small. The configuration inputs and information outputs of the RTP encode unit are all non blocking.

At the receiver side, when a valid RTP packet is received, the RTP timestamp from the packet is sent to the time difference unit, see "Time Diff" in Fig. 5.7. This timing channel counts the local clock cycles and everytime a new timestamp is received, it calculates the relative time difference. The jitter module blocks at the input until a new time difference is available. Using the variation of the relative time difference, it estimates the network jitter. Furthermore, after each received valid RTP packet, the clock recovery gets the current filling level of the jitter buffer. Several filling levels are averaged to filter out the network jitter. If the averaged values show an increase or decrease of the buffer filling level, the frequency of the numerical controlled oscillator (NCO) is adapted accordingly.

The recovered clock signal is used to control the blocking output of the G711 decode module. In each clock cycle, a decoded sample for each channel is written to the FIFO. After that, the G711 module reads new samples from the jitter buffer, decodes them by using a table, stored in a Const Array, and blocks until the next clock cycle. The FIFO channel between the G711 module and the resampler has a second clock input, connected to the codec clock. At the end of each cycle of the codec clock, the state of the FIFO is stored. So that the resampler can later read all values stored at this time instant. The resampler itself blocks at the output. Thus, after each codec clock cycle, the resampler starts reading all FIFO values, which were stored at the time of the clock edge. The number of values depends on the phase and frequency difference of the recovered clock from the VoIP opponent and the local codec clock. In the resampler, the samples are stored in a Deque, which is used as a shift register. The current phase difference, which is needed to read the appropriate coefficients from the Const Array, is measured in the phase difference timing channel, see "Phase Diff" in Fig. 5.7. Therefore, the entire receiver consists of two pure communication channels and three timing channels.

The designed model is first simulated at the system level. Therefore, a simple sine signal is used as source and RTP packets are generated via the RTP transmitter. The Local Link interfaces of the transmitter and the receiver are connected via a channel, which adds a network delay and jitter. To test both the clock recovery and the resampling, the receiver is operated with a different data clock than the transmitter. For this high level simulation abstract models of all timing and communication channels are used. The CTL elements are connected to their high level implementation. All template data types are set accordingly, so that the calculations are performed with maximum precision. Each module has been tested before. This first system simulation serves as a verification of the overall system functionality and it should point out errors in the interaction of the single modules.

5.2.3 G711 Audio Decompression Designed as HWSW-Module

In this Section, a code example of one of the HWSW-Modules implemented in the RTP engine is presented. The shown example is the G711 decode module, which is the simplest of the RTP engine. Lst. 5.1 shows the complete module implementation. Although the algorithm is very simple, it represents a good example, since it possesses all typical elements of a HWSW-Module and it also uses an element of the CTL library.

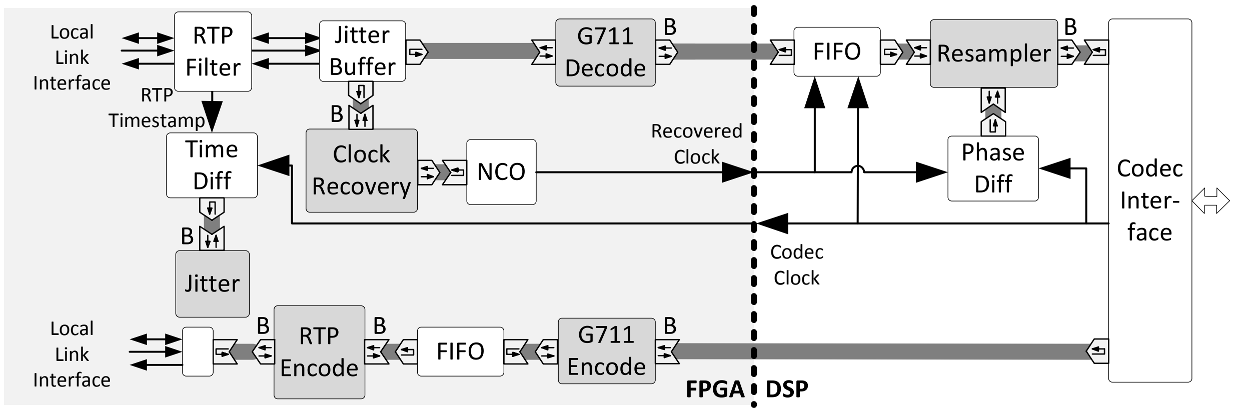
The module has one input and one output. Both ports are realized using the serial interface. To transport the samples of the left and the right audio channel via one port, the interface is defined to use `TSamples` objects, see line 5 and 6 in Lst. 5.1. `TSamples` is a simple C++ structure, which holds the samples of the left and the right audio channel. The actual data type of the audio samples can be defined via a template parameter. Whereby, different parameters are used for the input and for the output, since the A-Law input has eight bits and the PCM output has 13 bits.

The computation in the module itself is very simple. In principle, an A-Law input value is used as index to read the corresponding PCM output value out of a table, which is realized using the CTL element Const Array. Using the template parameters, it is possible to transform the HWSW-Module to a synthesizable hardware module or to a pure software module.

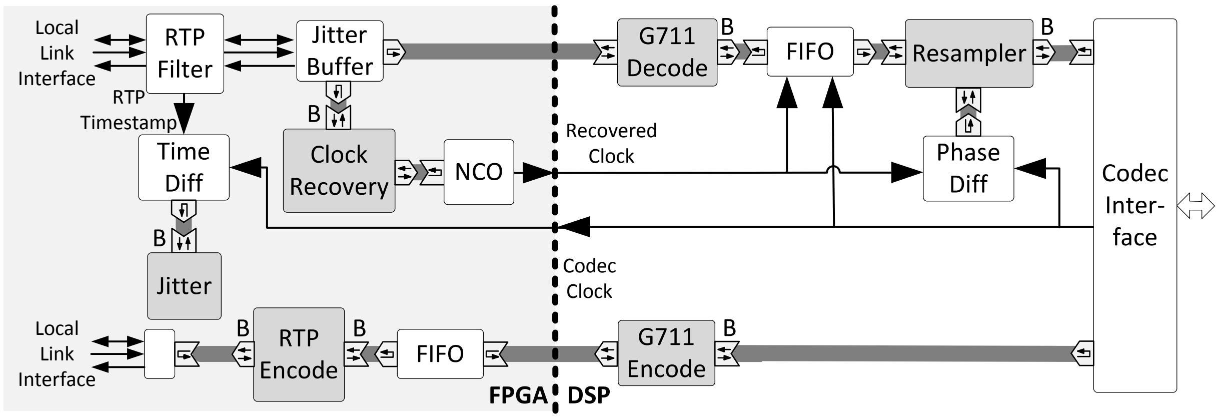
Listing 5.1: The `G711Decode` HWSW-Module as a simple example of one of the modules of the RTP engine. The A-Law coded intput is used as index to read the uncoded PCM values from the table `PCMTable`

```
 1  template <typename TDataIn, typename TDataOut, typename TContainer>
 2  SC_MODULE(G711Decode) {
 3    HWSW_MODULE
 4
 5    sc_port<Serial_Read_IF <TSamples<TDataIn> > >    iALaw;
 6    sc_port<Serial_Write_IF <TSamples<TDataOut> > > oPCM;
 7
 8    SC_CTOR(G711Decode) {
 9      HWSW_THREAD(Convert);
10    }
11
12          void Convert() {
13      {
14        END_RESET;
15      }
16
17      while(1) {
18        //read input
19        TSamples<TDataIn> ALaw = iALaw->read();
20
21        TSamples<TDataOut> PCM;
22        //conversion from alaw to linear pcm
23        PCM.Right = PCMTable.read(ALaw.Right);
24        PCM.Left  = PCMTable.read(ALaw.Left);
25
26        //write output
27        oPCM->write(PCM);
28      }
29    }
30
31    ctl::const_array<TContainer> PCMTable;
32  };
```

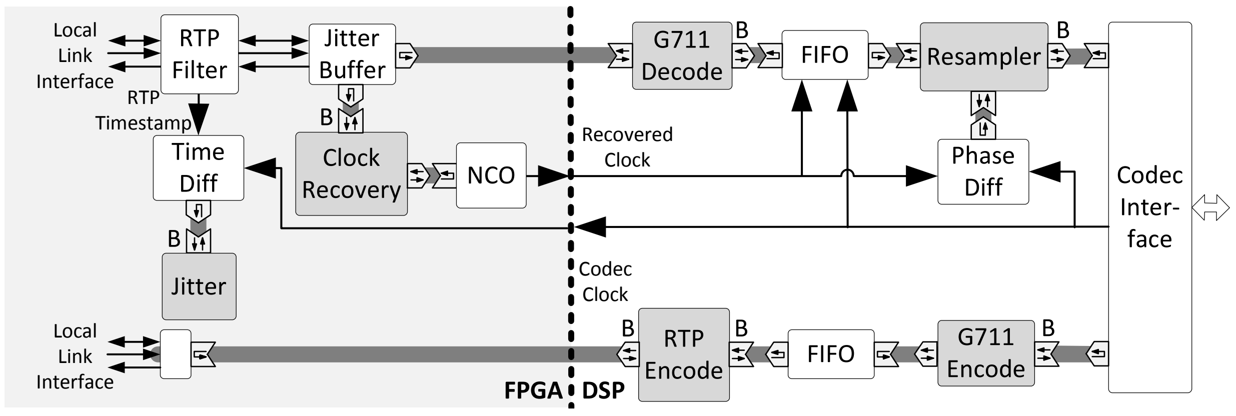
5.2.4 Different Hardware/Software Solutions



(a) HW/SW partitioning 1: one HWSW-Module switched to SW.



(b) HW/SW partitioning 2: three HWSW-Module switched to SW.



(c) HW/SW partitioning 3: four HWSW-Module switched to SW.

Figure 5.8: Different HW/SW partitionings of the RTP engine. Starting with the first partitioning, where all modules except the resampler are realized in hardware, more and more HWSW-Modules are switched to software.

In this Section different HW/SW partitionings of the RTP engine are presented. For each partitioning it is important to consider the complexity of the HW/SW interface. The communication effort between FPGA and DSP should be kept low to reduce the interface design effort. Furthermore, each data exchange is a time consuming process. Thus, the more data is exchanged, the more difficult it is to fulfill system timing requirements.

The first possible partitioning is the pure hardware solution, where each task is realized on the FPGA. This is of course a solution, which stands to reason, since an FPGA is used anyway in the VoIP system. In this case, the RTP engine is designed as synchronous design clocked by an oscillator with a frequency in the multi-digit MHz range. The codec clock and the recovered clock would be synchronous to this system clock. The timing channels can all be realized using clock cycle counters. In the time difference unit the codec clock's cycles are counted and compared to the received timestamps. The phase difference unit connected to the resampler, measures the phase difference of the clock signals with the system clock's resolution and the NCO can as well be implemented using a counter. After achieving a certain count value, the generated clock signal is inverted. This value corresponds to half of the period of the clock signal and is controlled by the clock recovery module. The pure hardware solution has also the advantage that the parallelism in hardware significantly simplifies the fulfillment of timing requirements. However, outsourcing some modules to a DSP might enable a switch to a smaller FPGA and further reduces costs and power consumption.

Since the Ethernet interface is realized on the FPGA anyway, it is reasonable to start shifting modules to the DSP at the codec side to keep the communication effort low. The first module at this side would be the codec interface itself. However, the implementation of only a codec interface would not justify the deployment of a DSP. Therefore, the first reasonable HW/SW partitioning is if the resampler is realized in software. Thereby, it is possible to realize the connected channels, the FIFO and the phase difference timing channel, on the DSP as well or they can be realized on the FPGA. However, it is not possible to separate them, since it is important for the correct behavior of the resampler that the codec clock arrives at the FIFO output at the same time as at the input of the phase difference unit. This can only be ensured if both modules are implemented in hardware in parallel or if the code fragments are executed immediately after each other and their execution cannot be interrupted. Fig. 5.8(a) shows this first HW/SW partitioning, whereby in this case, the FIFO and the phase difference unit have been mapped to the DSP. To realize the phase difference measurement in software accurately enough, a hardware timer or a real-time clock has to be used.

In Fig. 5.8 two other solutions are shown. Fig. 5.8(b) shows the RTP engine with the G711 decode and encode modules mapped to the DSP. It is indeed possible to realize one of these modules on the FPGA and the other one on the DSP. In the third solution 5.8(c) also the RTP encode unit is moved to the software side. Another alternative would be to move the clock recovery to the DSP. The NCO would require the use of an additional hardware timer. However, the realization of the NCO on the DSP would decrease timing accuracy. Timing tasks like the NCO or the phase measurement cannot be realized in software as accurate as in hardware because of the lack of parallelism. This leads to additional noise in the system. The inaccuracy corresponds to jitter, which is added to the clock signal. Therefore, mapping the NCO to software would significantly reduce system quality.

The accuracy is not a problem if the jitter module is moved to the DSP. The time difference unit only has to count codec clock cycles. Due to the low frequency of 8 kHz this can be implemented with sufficient accuracy in software. However, it would increase the communication effort. The RTP timestamp has to be transmitted to the DSP and the calculated jitter has to be transmitted back to the FPGA. Since the jitter module is very simple, shifting the jitter to software would not significantly change the required hardware resources. Accordingly, it is reasonable to realize the jitter module on the FPGA.

5.2.5 The Hardware/Software Interface

The three different alternative partitionings shown in Fig. 5.8 basically all have the same interface. They have a data interface at the receiver and the transmitter and two clock signals have to be exchanged between the chips. For the clock signals simple GPIOs can be used at the DSP side. To transmit the codec clock to the FPGA, the DSP can toggle a pin simultaneous to the clock cycles. At the FPGA side, the clock signal is then synchronized to the system clock. In the other direction, the FPGA can toggle a pin, which is connected to an input pin of the DSP. To react sufficiently fast enough, the DSP can be configured so that an interrupt is triggered on an edge, which occurs at the input.

Table 5.1: Minimum data rates for different HW/SW partitionings. The data rates are based on an audio sampling frequency of 8 kHz.

HW/SW Partitioning	Data rate
Partitioning 1	256 kbit/s
Partitioning 2	416 kbit/s
Partitioning 3	681 kbit/s

The data interfaces are very similar for the first and second partitioning. As well on the sender side as on the receiver side, single audio samples have to be transmitted over the interface. Using the first partitioning, uncoded samples with 13 bits are transmitted and in the second example 8 bit coded samples are transmitted. The timing requirements for the interfaces are very relaxed. Following the sampling rate of 8 kHz, one sample per channel has to be transmitted into both directions every 125 μs. This results in a required data rate of approximately 256 kbit/s for the first and 416 kbit/s for the second partitioning. In the third alternative, the RTP encode module is moved to the DSP. Therefore, whole RTP packets have to be transmitted over the HW/SW interface. With 10 samples per channel per packet, see Sec. 5.1.4, and the minimum Ethernet, IP, UDP and RTP headers follows a packet size of 74 bytes. A packet is sent every 1,25 ms, which results in a data rate of 681 kbit/s. Tab. 5.1 summarizes the data rates of all three partitionings shown in Fig. 5.8. Compared to modern high speed interfaces, these data rates are very small.

Typically, different interfaces are available on a DSP. TI DSPs for example usually have a serial high speed interface called McBSP [60] and the so-called external memory interface (EMIF) [62], which connects an external device to the address/data bus of the DSP. In the asynchronous mode, the DSP timing is adapted to the external device's timing by adding wait states. The development effort for the EMIF FPGA interface is small compared to the more complex McBSP interface, where a complex serial protocol has to be implemented. Using the EMIF, FPGA memory simply can be mapped into the address space of the DSP. The FPGA on the prototyping board is clocked with a 100 MHz clock. Under the pessimistic assumption of one cycle setup and one cycle hold time and another two cycles latency for the actual read or write operation, this would lead to an overall latency of 40 ns. Without considering turnaround times between the different read and write operations, this would lead to a data rate of 800 mbit/s for a 32 bit EMIF, which is a thousand times faster than required by the three partitioning solutions. This shows that a 32 bit external memory interface is definitely sufficient for this RTP engine design. Due to the lower development effort and due to the sufficient data rate, the EMIF interface is chosen as data interface between FPGA and DSP.

5.3 Prototyping Realizations

The Tripartite Design Approach simplifies the generation of a hardware or software implementation from the high level model. Thereby, low level tools can be used to estimate design parameters like hardware resources, performance and power consumption. Another important advantage of the design flow is the fast generation of prototypes. In this case study, the RTP engine has been designed using the TDA to find the best HW/SW partitioning for this part of the VoIP engine. In Sec. 5.2.4 already four different partitionings have been introduced. Now, for all four alternatives a functioning prototype should be realized. Therefore, first the used prototyping hardware is presented. Then the steps, required to transform the high level model to the different prototypes, are illustrated. Different realization and implementation details of the four prototypes are presented as well. Finally, results from different simple measurements and tests proof the general functionality of the prototype. Thereby, especially the end-to-end delay has been measured, which has been identified as an important design constraints to build a system appropriate for safety critical applications.

5.3.1 The Prototyping Set-up

In this Section, the basic prototyping set-up is presented. As already mentioned in Sec. 5.2.1, the architecture of the VoIP engine is already fixed to a Xilinx Virtex4 FPGA and probably a TI DSP. One reason for this decision was the availability of prototyping boards with the corresponding ICs.

The first board is the Xilinx ML405 [34]. It is a development board with a Virtex4 XC4VFX20 [65] and many different interfaces. Relevant for the realization of the RTP engine prototype is the RJ-45 connector for the Ethernet network interface and the audio jacks for microphone and headphones, see Fig. 5.9. The RJ-45 connector is connected to a PHY chip, the Marvell Alaska 88E1111 Gigabit Ethernet transceiver [35]. The PHY can be directly connected to the embedded Tri-Mode Ethernet MAC [68] in the FPGA. The audio interface is realized using the National Semiconductor LM4550 multi channel audio codec [30], which drives the audio jacks and provides an AC97 compliant digital interface to the FPGA. Another important feature of the prototyping board are the expansion headers, which allow a user defined connection. This connection is used to connect the FPGA to the DSP board. The ML405 has two crystal oscillator sockets, where one is populated with a 100 MHz clock oscillator, which can be used as the FPGA system clock signal.

The second board is the DSK6455 [61] with a Texas Instruments TMS320C6455 [36] DSP. The TMS320C6455 is a high performance fixed point DSP from TI. On this board it is running with a clock frequency of 1 GHz. As the FPGA board it has audio jacks connected to an audio codec chip. The used codec chip is the Texas Instruments TLV320AIC23 stereo audio codec [28], which is connected to the DSP via the McBSP interface. On board are three different expansion headers, see Fig. 5.10. This board and two ribbon cables can be used to connect the expansion headers of the DSP board with the expansion headers of the FPGA board. Thereby, an EMIF interface with up to 32 data and 19 address lines can be set up.

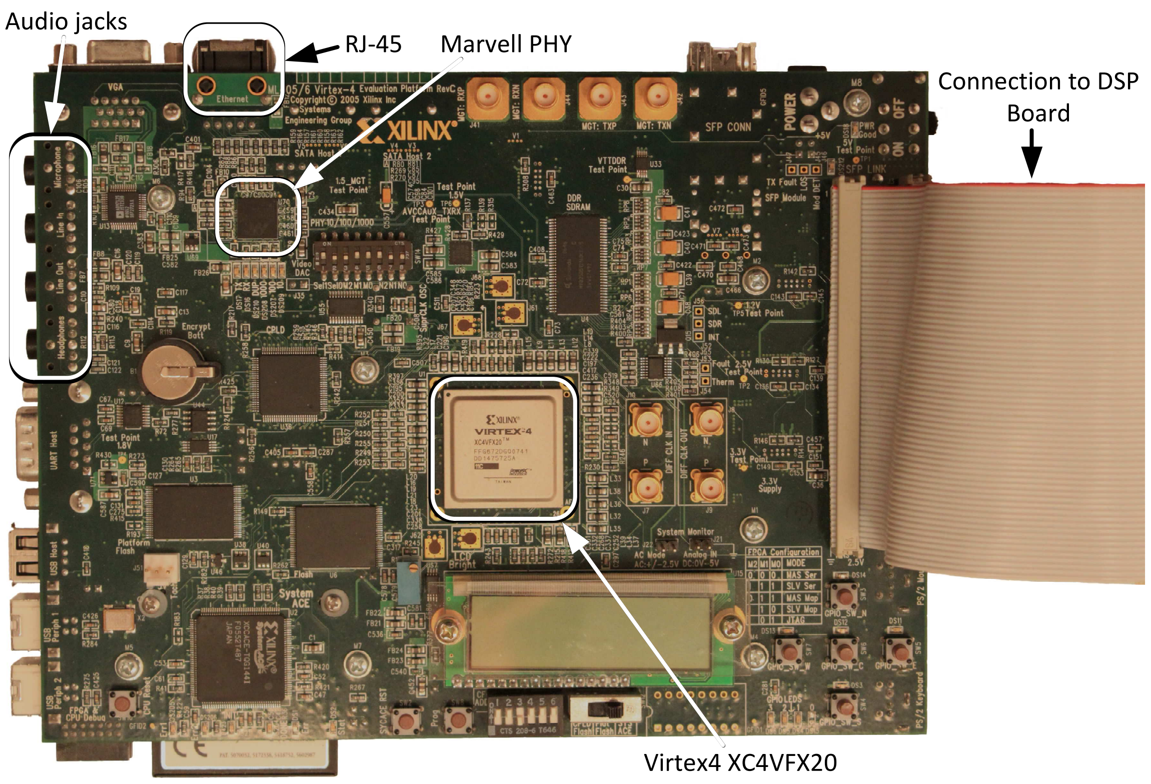


Figure 5.9: The ML405 FPGA prototyping board with the Xilinx Virtex4 XC4VFX20. Via expansion headers and a ribbon cable, the board is connected with the DSP board. Audio jacks, RJ-45 connector and the Ethernet PHY are highlighted as well.

There is no direct connection between the sampling clock of the audio codec and the DSP respectively the FPGA on both boards. In the first place, this clock is required by the audio resampler, see Sec. 5.2.2. Therefore, it is necessary to recover the sampling clock from the digital codec interface. Obviously, this adds additional jitter to the clock signal, which further increases the noise, which is added by the resampler. For this case study the audio quality is still sufficient. However, for a real product a more sophisticated solution has to be found.

5.3.2 Prototype Realizations

In this Section the four realized and tested prototypes of the RTP engine are presented. One of the four prototypes is one chip FPGA solution, while the other three prototypes are mixed FPGA/DSP realizations. In the following are the steps required to transform the high level model to an actual prototype briefly presented. The most important implementation details of all four prototypes are illustrated as well.

5.3.2.1 One-Chip FPGA Prototype

In the pure hardware solution, the whole RTP engine is realized on the ML405 FPGA prototyping board. The HWSW-Modules can be used directly. The code of these modules does not have to be changed. The template parameters of the data types are set to hardware data types like `sc_uint`

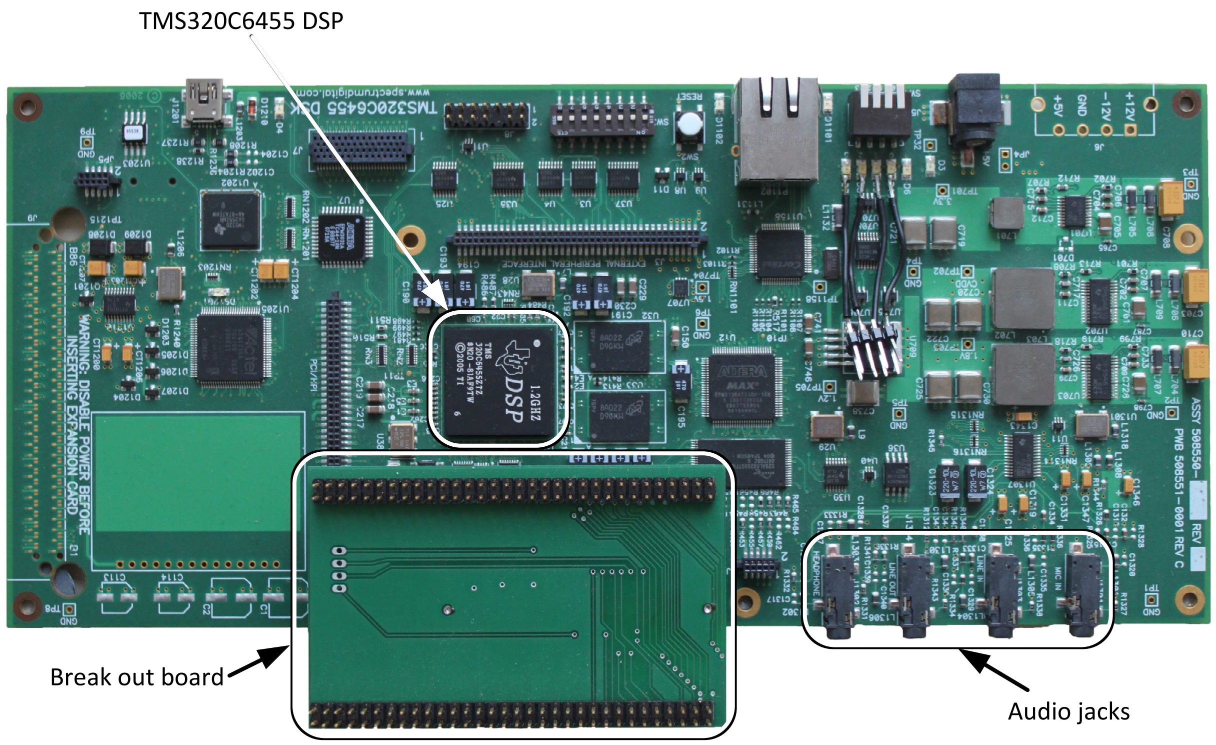


Figure 5.10: The DSK6455 DSP prototyping board with TI TMS320C6455 DSP. In addition, the four audio jacks and the break out board are emphasized.

with arbitrary bit width. For most parts in the design, the bit width is fixed because of the used network protocol. However, the bit width of the phase difference measurement for example can be adjusted freely. Via simulation different bit width can be tested to find the best accuracy and hardware complexity trade off.

The implementations of the CTL elements are replaced with the corresponding synthesizable hardware implementation, which has to be connected to a memory structure. In the RTP engine most CTL elements have different bit width. Therefore, it is not possible with the current CTL version to map different data structures to the same memory structure. Next, the communication and timing channels have to be replaced with synthesizable hardware channels. All timing channels, the NCO, the time difference and the phase difference unit, are implemented using counters. The high level codec interface is replaced with a synthesizable AC97 compliant hardware interface.

To form a synthesizable SystemC module, each HWSW-Module is instantiated in a top level module together with the surrounding communication and timing channels. Since, the communication channels typically connect two HWSW-Modules, they are replaced by two adapters, which translate the high level function call interface of the HWSW-Module to the low level signal interface of the top level module. Thereby, the typical synthesizable module with a cycle accurate interface description and untimed computation is built.

An example for such a top level module is shown in Fig. 5.11. For the resampler two memory structures and three channels are required. The memory structures are connected to the Const Array and to the Deque. The output is connected to general blocking output adapter. The phase difference unit and the FIFO are application specific elements, which also basically translate the

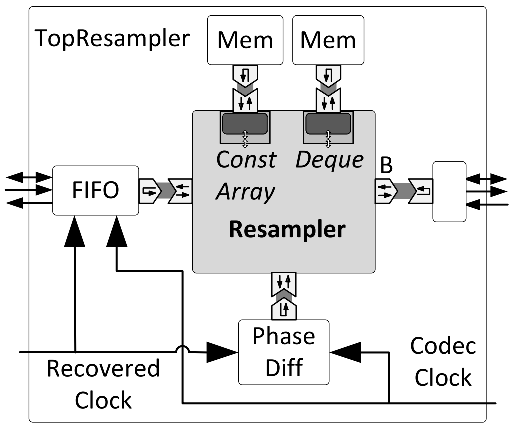


Figure 5.11: The synthesizable module `TopResampler`, which instantiates the HWSW-Module Resampler and the surrounding communication and timing channels.

function interface to a low level signal interface.

The refined RTP engine model consists of different synthesizable hardware modules. This model has then been simulated using a 100 MHz clock and reset signal. By this simulation, the digital communication between the modules and the usage of the actual hardware data structures can be verified. After the successful hardware simulation, each module has been synthesized for a Virtex4 using the ForteDS Cynthesizer. The generated Verilog code has been simulated again using the same SystemC test bench. This simulation already provides information about the latency of each module and thereby, it can be verified that the RTP engine fulfills the given timing requirements. Finally, the Verilog code has been further synthesized using the logic synthesis tool from Xilinx, the XST, to generate the bit file, which is used to configure the FPGA.

5.3.2.2 HW/SW Prototypes

Based on the HW prototype, for the HW/SW prototypes different parts are moved to the software domain. Thus, the AC97 codec interface is not required anymore. Instead, the external memory interface is inserted. It basically consists of two dual port block RAMs. One is used for the RX path of the prototype and one for the TX path. Both RAMs are mapped into the address space of the DSP. Whereby, the most significant bit decides which RAM is addressed. The second port of both block RAMs is connected to the RTP engine. This EMIF unit can be used in each presented HW/SW partitioning. Only a small adapter is required, which translates the signal interface of the connected module to the block RAM interface. Like presented in Sec. 5.2.5, this signal interface is different for each HW/SW partitioning, e.g. in the first alternative, see Fig. 5.8, 13 bit samples have to be transported via the HW/SW interface, while in the second alternative only 8 bit samples are exchanged between FPGA and DSP.

The modules, which are moved from the hardware to the software domain, are simulated using high level channels and high level container implementations. Via an abstract channel, these modules read and write data from and to the EMIF from the DSP side. By this simulation, it is possible to verify timing and functionality of the hardware part of the prototype. The software part is verified directly on chip using a JTAG debugger. Thereby, the DSP interact already with the configured FPGA. Of course, a simulation of the whole HW/SW prototype would be possible by using an instruction set simulator (ISS). Since an ISS, which could easily be integrated into a SystemC simulation, was not available, the in system debugging method has been chosen.

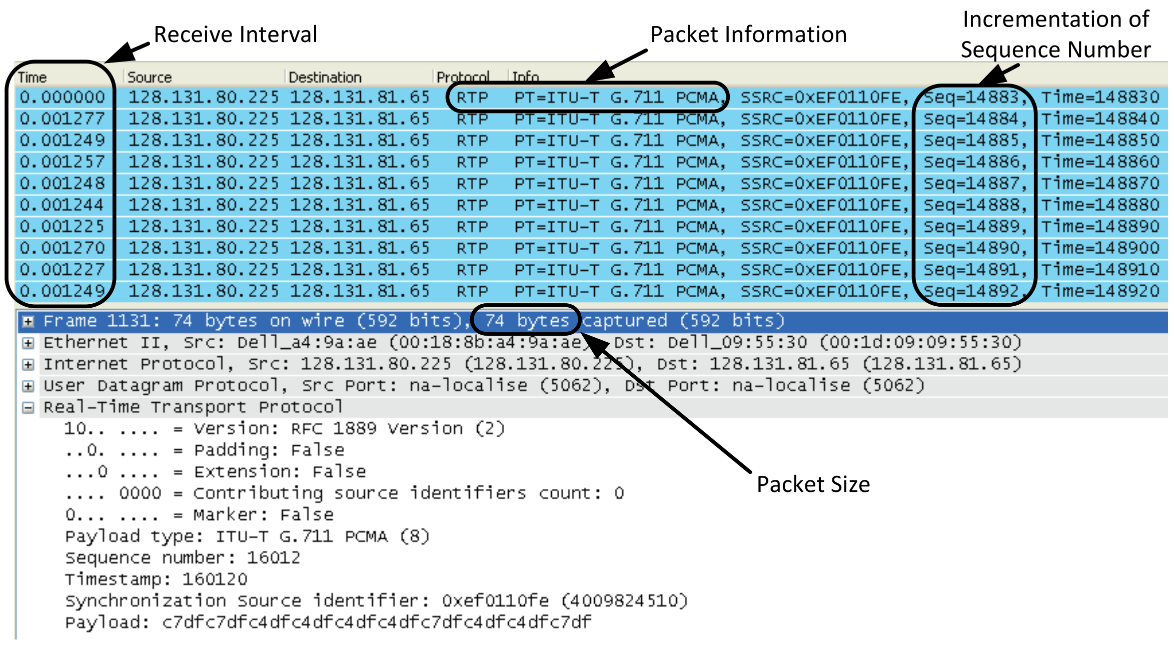
The HWSW-Modules, which are moved to the DSP, can be used without any changes. All CTL container adapters are connected to pure software implementations. Currently, the library provides two alternative software implementations: One is based on STL elements and therefore uses dynamic memory management and the other is a fully static implementation. in Sec. 5.3.3 both alternative implementations are compared concerning their performance. Via template parameter, the data types are set to software data types, typically with 8, 16 or 32 bit width. Each HWSW-Module is executed as a separated thread in the realtime operating system DSP/Bios from Texas Instruments. Thereby, all modules run virtual in parallel and they are blocked in the communication channels using semaphores.

The only timing channel implemented on the DSP, is the phase difference unit, which is realized using a hardware timer. This timer is reset after each clock cycle of the recovered clock. After each clock cycle of the codec clock, the timer value is read out. It represents the current phase difference of the two clock signals. The DSP board does not provide the possibility to use the codec clock directly. Therefore, it has to be regenerated via the digital codec interface as well. Since the audio codec and the DSP are connected via the McBSP interface, the McBSP receive interrupt can be used as codec clock. Each time the codec sends new audio samples to the DSP, an McBSP receive interrupt is caused. In the interrupt service routine, a GPIO pin is toggled to transmit the clock signal to the FPGA. Further, a function in the FIFO at the resampler's input is called to store the current FIFO status. Via a semaphore, the blocked read operation of the resampler is released and the resampler reads all samples, which have been in the FIFO, when the McBSP interrupt has been triggered. The recovered clock is realized using an external interrupt. The FPGA toggles a pin after each clock cycle of the recovered clock and thereby causes an interrupt at the DSP.

5.3.3 Tests and Measurements

All four different prototypes have been tested using simple test scenarios. Further, measurements have been made to check the fulfillment of design constraints and quantify certain quality parameters. To verify the general functionality, consecutively different prototypes have been connected directly via a one-on-one Ethernet connection. All configuration parameters like the IP addresses have been set statically. In this way, all the call setup procedure, which would be realized in the stack implementation part, is not necessary. It was possible to establish a working bidirectional voice link for all prototypes. The clock recovery adapted the clock frequency to avoid buffer overflows and underruns. The resampling process correctly converted the audio samples from the sender's to the receiver's sampling domain, so that a glitch free audio transmission could be performed. This test shows that the concept of a small jitter buffer combined with the clock recovery and the resampler in principle works.

The correct behavior of the sender is further measured with the network sniffer Wireshark. Fig. 5.12 shows a screenshot of a packet trace of the packets sent by the pure hardware prototype. The trace shows that the received packets are recognized as correct RTP packets using the G.711 A-Law audio codec. The column named Time illustrates the time span between the reception of two consecutive packets. It can be seen that the packets are received in an approximate interval of 1.25 ms, which corresponds to the expected behavior. The packet trace further shows the correct incrementation of the sequence number by one and the timestamp by ten for



```
Time      Source         Destination    Protocol Info
0.000000  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14883, Time=148830
0.001277  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14884, Time=148840
0.001249  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14885, Time=148850
0.001257  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14886, Time=148860
0.001248  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14887, Time=148870
0.001244  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14888, Time=148880
0.001225  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14889, Time=148890
0.001270  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14890, Time=148900
0.001227  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14891, Time=148910
0.001249  128.131.80.225 128.131.81.65  RTP   PT=ITU-T G.711 PCMA, SSRC=0xEF0110FE, Seq=14892, Time=148920
⊞ Frame 1131: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
⊞ Ethernet II, Src: Dell_a4:9a:ae (00:18:8b:a4:9a:ae), Dst: Dell_09:55:30 (00:1d:09:09:55:30)
⊞ Internet Protocol, Src: 128.131.80.225 (128.131.80.225), Dst: 128.131.81.65 (128.131.81.65)
⊞ User Datagram Protocol, Src Port: na-localise (5062), Dst Port: na-localise (5062)
⊟ Real-Time Transport Protocol
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    Payload type: ITU-T G.711 PCMA (8)
    Sequence number: 16012
    Timestamp: 160120
    Synchronization Source identifier: 0xef0110fe (4009824510)
    Payload: c7dfc7dfc4dfc4dfc4dfc4dfc7dfc4dfc4dfc7df
```

Figure 5.12: Network trace of a VoIP call using Wireshark. Only received packets are shown. Different parameters showing the correct behavior of the VoIP engine are highlighted.

each packet. The trace shows the detailed decoding of one RTP packet. Additionally, the trace displays one packet in detail. This enables the identification of the single fields of the RTP packet. The details show that the packet size corresponds to the theoretically calculated packet size of 74 bytes.

Table 5.2: Expected and measured end-to-end delay of the RTP prototypes. The expected delay is broken down into the components causing the delay.

Delay source	Delay
RTP encode	1.25 ms
Jitter buffer	1.25 ms
Resampler	1 ms
Audio codec	3 ms
Expected end-to-end delay	6.5 ms
Measured end-to-end delay	7 ms

In Sec. 5.1.1 the end-to-end delay has been identified as an important design constraints for a VoIP engine targeting safety critical applications. Therefore, measurements are performed to analyze the end-to-end delay and further to evaluate the benefit of the design decisions, which have been made to minimize the delay. Tab. 5.2 lists the different major delay sources in the design and the estimated delay. The first listed delay source is the RTP encode unit, which has to collect 10 samples before the first packet is sent. With a sampling rate of 8 kHz it takes 1.25 ms to collect 10 audio samples. At the receiver side, two RTP packets have to be received until the jitter buffer is half filled. Only then, the receiver starts to read out samples from the jitter buffer. This adds an additional delay of 1.25 ms. Another delay source is the resampler. According to Sec. 5.1.3.2 it has a delay of 1 ms. The final delay source in the Table is the audio codec, which adds a delay of approximately 3 ms. This delay may be caused mainly by filtering processes in the audio codec chip. It has been estimated by measuring the delay of an audio signal, which is converted to digital and then directly converted back to analog. The sum of all these delays results in an expected minimum end-to-end delay of 6.5 ms. However the actual delay will be higher because of additional buffering of e.g. Ethernet packets in the design. The end-to-end delay has then been measured several times using each of the prototypes. From these measurements an average delay of around 7 ms has been calculated for all prototypes. This is far below the delay of a conventional VoIP engine, where the jitter buffer alone produces a delay of 100 ms [WCHL09].

Another important design goal is of course the audio quality. The concept with the clock recovery and the resampler enables a glitch free audio conversion with a low latency. However, it reduces the audio quality because of the additional filtering process. To evaluate the audio quality of the prototypes different measurements have been performed. Therefore, a sine signal with 1020 Hz has been transmitted using the RTP engine. First, the transmission has been performed using the high level simulation. In this case, the output signal of the receiver is written to an audio file, which is later analyzed using Matlab [14]. The same sine signal is played back via the sound card of a PC. The analog audio signal is then used as input signal at one of the prototypes, which transmits the signal via Ethernet as RTP stream. At the receiver, the signal is converted back to an analog signal, which is then recorded by using the PC.

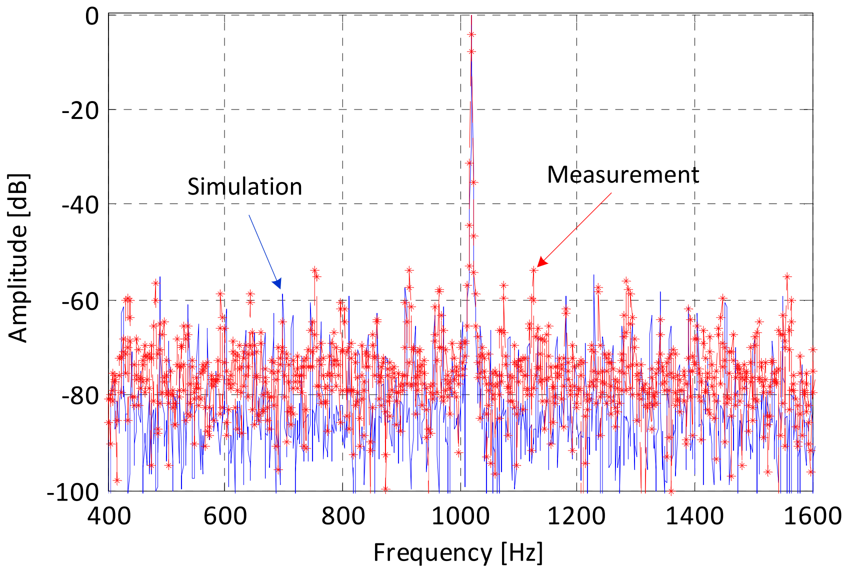


Figure 5.13: Frequency spectrum of the functional simulation compared to a measurement using the pure HW prototype. The test signal is a 1020 Hz sine signal and the spectrum is normalized to 0 dB. The general noise floor is increased in the measurement, which can be explained by not simulated noise sources.

In Fig. 5.13 frequency spectrum of the signal generated via simulation and the spectrum of the measured signal of the pure HW prototype is shown. The spectrum has been generated using a Hanning window [Nut81] and a 32 k point Fast Fourier Transform (FFT) in Matlab. It can be seen that there is almost no difference considering the maximum distortion, which is taken into account in the spurious free dynamic range (SFDR). According to [BR01, p. 298] a spur is defined as: "Any nonsignal component that is confined to a single frequency." The SFDR is then defined as the difference of the fundamental signal and the maximum spur in the frequency domain. In Tab. 5.3 the SFDR for different signals is shown. The SFDR of the signal generated by a simulated transmission and the signal generated by a transmission via the hardware prototype is only 0.04 dB. However, Fig. 5.13 shows also that the general noise floor is higher in the measured signal. A reason therefore might be that not all noise sources have been simulated. The simulated system uses an ideal codec clock, while the prototype has to regenerate this clock signal from the digital codec interface. The rise of the general noise floor is reflected in the Signal-to-Noise and distortion ratio (SINAD), see Tab. 5.3. It is 1.66 dB higher in the measured signal. Different definitions exist for the calculation of the SINAD. In this work it is defined as the ratio of the fundamental signal power compared with the power of the noise and all distortions, whereby the direct current (DC) is excluded [BR01, p. 281].

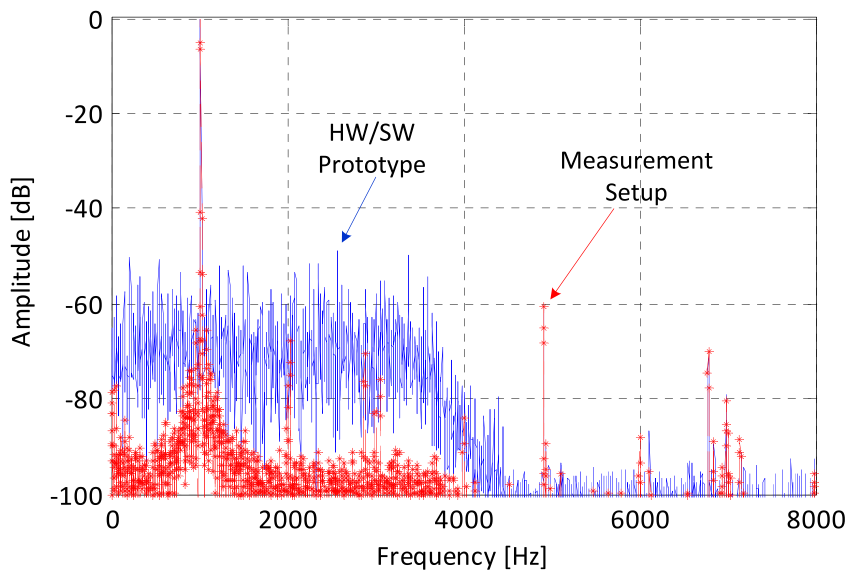


Figure 5.14: Frequency spectrum of the measurement setup compared to a measurement of one of the HW/SW prototypes. The test signal is a 1020 Hz sine signal and the spectrum is normalized to 0 dB. The spectrum of the measurement setup shows several spurs, which result from non-idealities on the prototyping board or on the PC sound card.

To illustrate distortions which result from the measurement setup and from the prototyping boards, the measurement setup has been measured without the RTP engine. This has been performed by configuring the FPGA, so that it returns the recorded digital audio signal directly back to the audio codec. Fig. 5.14 shows the measurement of one of the mixed HW/SW prototypes compared with the measurement setup. Of course, the noise floor of the measurement setup is lower compared to the prototype. However, the Figure also shows that there are several spurs in the spectrum of the measurement setup, which result from non-idealities on the prototyping boards or on the PC sound card.

Table 5.3: Audio quality key figures of an ideal A-Law coded signal and of the signals generated via simulation and prototype measurement. The measurements show that SFDR and SINAD values are in the range of an ideal A-Law coded signal.

	SFDR	SINAD
Ideal A-Law	51.3 dB	38.18 dB
Simulation	52.8 dB	39.28 dB
HW Prototype	52.76 dB	37.62 dB
HW/SW Prototype	48.67 dB	35.04 dB

Another significant source of distortions in the system is the non-linear quantization by the G.711 A-Law audio codec. Due to the non-linear characteristic, the quantization error is higher for signal parts with a higher amplitude compared to signal parts with a lower amplitude. The reason is that voice consists of more parts with a lower amplitude compared to parts with a higher amplitude. In Tab. 5.3 an ideal A-Law coded sine signal is compared to the RTP engine simulation and to the measurements. All signals have been sampled with a sampling rate of 8 kHz. Then a 32 k point FFT has been performed to transform the signal to the frequency domain. For the ideal signal and for the simulation, a signal with the same amplitude has been used. Due to the lack of appropriate equipment, the amplitude for the measurements only has been calibrated approximately.

The worst SFDR and SINAD has been calculated for the measurement of the HW/SW prototype. Its SINAD is 3 dB below the ideal A-Law signal. The signal generated by the simulation has even a better SINAD compared to the ideal A-Law signal. A reason for that might be the filter in the resampler, which starts attenuating with a frequency of 3.6 kHz, see Sec. 5.1.3.2 and thereby reduces the overall noise power in the signal.

In general, the measurements show SFDR and SINAD values in the range of the ideal A-Law signal, which further shows that the concept using a clock recovery and the resampler can be deployed without a significant quality reduction. For a more expressive testimony about the audio quality of the prototype, more accurate and detailed measurements are necessary. The other performed tests are not sufficient to attest full correct functionality as well. However, for a first prototype series and for the purpose of exploring the design space, the results sufficiently prove the generation of working prototypes with acceptable audio quality. Further, with an average delay of 7 ms for all prototypes, the design goal to reduce the end-to-end delay to a minimum has definitely been achieved. This delay is significantly below conventional VoIP engines.

5.4 Design Space Exploration

In Sec. 5.1.1 four design goals have been identified: minimize power, costs, the end-to-end delay and the use of COTS components. During the architecture design one FPGA and an optional DSP have been chosen. In Sec. 5.3.3 measurements have shown that the design goal of a small end-to-end delay has been achieved. Thus, only the minimization of power and costs is missing. Therefore, a design space exploration is performed. The TDA enables the easy shifting of computation modules from hardware to software and vice versa. It simplifies the generation of different HW/SW implementations. Further, it provides the possibility to map data structures in hardware designs to either block RAMs (BRAM) or to registers and also to map different data structures to the same memory structure. All these possibilities are exploited now to compare different design solutions. Obviously, the generated implementations are not optimized. However, they can easily be analyzed using low level tools to get estimates of power and costs.

In the following, different hardware and software alternatives of the RTP engine are analyzed. Thereby, the area requirements for the FPGA and performance requirements for the DSP are derived. The selection of the actual FPGA and DSP type heavily influences the costs and the power consumption. The objective is to find the cheapest, most power efficient devices, which still fulfill the requirements. Then, costs and power consumption are estimated for all selected devices and for all different design solutions. Therefore, the power consumption is estimated via power analyzing tools provided by the chip vendors. The costs are calculated by taking typical prices from the Internet. Finally, the best solutions are identified using the calculated prices and the estimated power consumption.

5.4.1 Hardware Design Solutions

In this Section the hardware parts of the different partitionings are analyzed. To estimate the actual amount of hardware resources for the VoIP engine, the number of resources required for the stack implementation are taken into account as well. In Sec 5.2.1 it has been defined that a Virtex4 FPGA is used. The primary reason is the availability of a Xilinx prototyping board and a stack implementation, which also targets a Virtex4 FPGA. Further, Virtex5 and newer FPGA technologies turned out to be too expensive. Additional requirements are the availability of an embedded PowerPC and an embedded MAC to efficiently realize the Ethernet connection and the stack implementation.

Table 5.4: Estimated hardware resources in terms of registers, slices, lookup tables and block RAMs of the design components, which are in any case realized on the FPGA. The total number of resources corresponds to the minimum requirements for the FPGA selection.

	Registers	Slices	LUTs	BRAMs
PowerPC	6034	5235	6561	33
Ethernet MAC	445	331	478	4
RTP filter	396	338	539	0
Jitter buffer	88	159	305	1
Total	6963	6063	7 83	38

To estimate the minimum number of hardware resources, design components, which are in any case realized on the FPGA, are analyzed first. Tab. 5.4 lists the amount of different kinds of FPGA resources of the PowerPC, the Ethernet MAC, the RTP filter and the jitter buffer. The mentioned resource "slice" names a basic logic block in Xilinx FPGAs. It consists of two LUTs, two storage elements and multiplexers. More detailed information about the structure of an FPGA can be found in 2.3.1. The estimations for the PPC have been taken from [Wen11], while all other estimations in Tab. 5.4 have been generated by synthesizing the different design units with the Xilinx Synthesis Tool (XST). Therefore, the RTP filter and the jitter buffer first have been synthesized using the ForteDS Cynthesizer.

Table 5.5: Hardware resource and price information of the three smallest Virtex4 FX FPGA devices. Due to the minimum requirements, at least a Virtex4 XC4VFX20 has to be chosen. Prices have been taken from [48] (30/09/2011).

	Registers	Slices	LUTs	BRAMs	Price
XC4VFX12	10 944	5472	10 944	36	$ 154.80
XC4VFX20	17 088	8544	17 088	68	$ 329.84
XC4VFX40	37 248	18624	37 248	144	$ 526.68

Considering only the Virtex4 FPGAs with an embedded PowerPC and MAC core shrinks the selection to the FX class of the Virtex4 devices. The available hardware resources of the three smallest Virtex4 FX FPGAs are listed in Tab. 5.5. Additionally, a typical price, taken from [48] (30/09/2011), for each device is provided. Comparing the available resources with the minimum required resources, it turns out that at least the XC4VFX20 is required, especially because of the number of needed BRAMs and slices. Even when using the XC4VFX20, 70 % of the slices and 55 % of the BRAMs are in use. Since the next larger FPGA provides more than double the resources it can be assumed that the four different paritionings fit either in the XC4VFX20 or in the XC4VFX40.

To get an estimation of the required resources for all HWSW-Modules of the RTP engine, these SystemC modules have been synthesized using high level synthesis. The synthesis process has been directed to minimize the area. The reason is that performance is not so important, since the timing requirements are very relaxed because of the data rate of 8 kHz. Therefore, each generated hardware module requires several FPGA clock cycles to produce one output value. In the following the hardware requirements of the four different partitionings presented in Sec. 5.2.4 are analyzed. The question is primarily which solution can be realized on the smaller and significantly cheaper FPGA. The usage of CTL containers for the realization of data structures enables the possibility to simply compare different data structure to memory structure mappings. Whereby, the different bit width of all data structures in the design limit the mapping capabilities. It is only possible to map each data structure to a BRAM or to registers.

A look at the different modules and their data structures, makes clear that it is not useful to analyze all different mappings. The resampler, for example, has two different data structures: a Deque and a Const Array. However, the Const Array has 1002 entries and is therefore to large to map it efficiently to registers. This only would make sense if RAM blocks are very heavily used in the design. The same pertains to the G711 encode module. Its Const Array is with 4096 entries even larger and is therefore only mapped to BRAM. However, the decode unit has a smaller Const Array. Thus, the BRAM and the register alternative are analyzed. The RTP encode unit has two different data structures: a Queue and the protocol header fields. In this case all four mapping alternatives are analyzed.

The analysis shows that the crucial hardware resources are the slices. The number of Block RAMs available on the FPGA is sufficient for all partitioning of the RTP engine. Each variant of each module has been synthesized using the ForteDS Cynthesizer and the XST. The power consumption has then been estimated using the Xilinx XPower Analyzer, which analyses the generated netlist. Detailed information about the resource requirements of the different alternatives can be found in App. B.2.

Fig. 5.15 shows area versus power of 16 different alternatives of the hardware part of the RTP engine. The underlying data of Fig. 5.15 can be found in Tab. B.2. In this case area corresponds to the number of FPGA slices. The 16 different solutions result from the four different partitionings with different memory structure mappings. The dashed line in Fig. 5.15 indicates 3309 slices. This number results from the available slices on the XC4VFX20 subtracted by the number of slices used by the PowerPC and its peripherals. It corresponds to the maximum number of slices the RTP engine can have so that it still fits on the cheaper XC4VFX20. All design solutions above this line require the more expensive and significantly larger XC4VFX40 FPGA. Only the third partitioning (HWSW3) leads to a solution, which can be realized on the smaller FPGA, see ① in Fig. 5.15. The other partitionings have to be realized on the larger FPGA independent of the chosen data structure to memory structure mapping. Thus, the optimization focus of these alternatives can be on reducing the power consumption.

Another interesting realization possibility is solution number ②. It is a pure hardware solution with all data structures mapped to registers. Obviously, the required area (7866 slices) is very large. However it fits on the XC4VFX40 and it is the pure hardware solution with the least power consumption. The solutions for the partitioning two (HWSW2) and three (HWSW3)

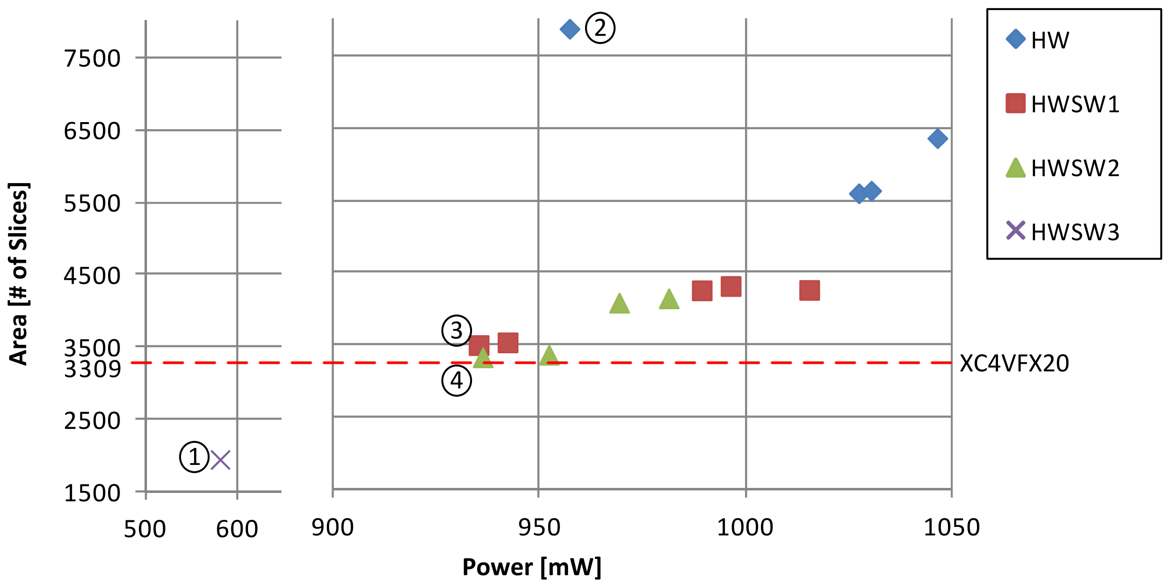


Figure 5.15: Area and power consumption of 16 different design solutions. HW denotes pure hardware solutions and HWSW1 to HWSW3 denote the mixed HW/SW solutions. Area is measured using the number of slices. The used block RAMs are not considered. The underlying data can be found in Tab. B.2

have comparable hardware resource requirements. The best solutions are ③ and ④, where all mentioned data structures are realized using BRAMs. Both solutions are slightly to large for the XC4VFX20.

5.4.2 Software Analysis

In the last Section only the hardware part of the different solutions has been analyzed. However, to completely evaluate the different realization possibilities of the RTP engine, it is necessary to analyze the software part as well. The best partitioning can only be found by comparing the estimated overall costs and power consumption of the different alternatives. To estimate the power consumption of a DSP the utilization of the CPU and of the different peripherals has to be measured up. Texas Instruments provides a power estimation spreadsheet for each processor, which calculates a power consumption estimate based on these utilization values, see e.g. [59] for a spreadsheet for the TMS320C6455 DSP. In the following the software part of the three in Sec. 5.2.4 mixed HW/SW solutions is analyzed. For each solution, four different possible DSPs are chosen and their estimated power consumption is calculated.

To estimate the utilization of the CPU, the runtime of each task is measured. This measurement could be performed using a cycle accurate instruction set simulator (ISS). However, due to the lack of appropriate ISSs and since the tripartite approach simplifies the generation of actual prototypes, the measurements are done using the prototyping hardware presented in Sec. 5.3.1. The runtime of a single task is measured by switching a GPIO on and off at the beginning and at the end of a thread. Thereby, it is possible to measure the execution time by using a logic analyzer. All measurement results are shown in Tab. 5.6

For the G711 encode and Decode unit, the measured time corresponds to the time required to read a value, decode or encode it and write the result. The measured runtime is very short, which is explained by the simplicity of the coding process, which almost only consists of reading the coded value from a table. In the case of the resampler the time from reading the input values to writing a new output value is measured. This measurement has been performed using different CTL implementations. One implementation is based on STL containers and uses dynamic memory management and the other uses a static implementation. As expected, the static implementation is faster and more efficient. Obviously, the static way is better suited for this case study, since a shorter runtime means less utilization and therefore a lower power consumption. For the RTP encode unit the measured time does not cover collecting the samples. The measured time span starts with the calculation of the UDP checksum and includes the complete sending process of the RTP packet. Whereby, on the DSP side sending the RTP packet means writing it to the FPGA via the EMIF interface.

Table 5.6: Measured runtime of different tasks per 125 μs period. The resampler and the RTP encode task have been measured once using static and once using dynamic memory management.

Task	Time	
Encode G711	650 ns	
Decode G711	530 ns	
HWSW1 Idle	111.77 μs	
HWSW2 Idle	104.95 μs	
HWSW3 Idle	102.917 μs	
	dynamic Mem.	static Mem.
Resampler	9.23 μs	30.64 μs
RTP encode	20.33 μs	34.75 μs

Further, the execution time of the idle task in each period has been measured. One period is 125 μs long and results from the 8 kHz sampling rate. The time has been measured in a period in which no RTP packet has been sent. Otherwise, the idle time would be shorter, approximately by the measured RTP encode time. 125 μs minus the idle time leads to the time in which the CPU is active. This time in relation to the period corresponds to the temporal utilization of the CPU. To include the RTP encode time in the calculated utilization, it is averaged over ten periods, since every tenth period a packet has to be sent.

Under the assumption that halving the CPU frequency doubles the utilization, it is possible to estimate a minimum DSP clock frequency for each partitioning. Due to the averaging of the RTP encode time over ten periods, the minimum clock frequency would result in a sending process which takes several periods. This would further delay the sending of a packet and therefore increase the End-To-End delay. Thus, the minimum frequency is estimated pessimistically with a theoretical maximum of 80 % CPU utilization. The calculated minimum clock frequency for the first partitioning is 133 MHz, for the second partitioning it is 201 MHz and for the third it is 242 MHz.

To select appropriate DSPs several constraints are defined. In Sec. 5.2.1 it has been defined that TI DSPs are used. The three prototypes of the partitionings HWSW1, HWSW2 and HWSW3 require a minimum of 124 kB, 126 kB and 153 kB memory. Therefore, one constraint is a minimum of 256 kB of internal memory. Additionally, the DSP has to have a EMIF interface for the communication with the FPGA and a timer is required for the phase difference measurement of the resampler. For the connection with the audio codec, either a McBSP or a I2S interface is needed for the data interface and the control interface is realized using an I2C or an SPI interface.

The supported clock frequency should be low enough to keep the power consumption low and it should be higher than the calculated minimum frequencies.

For low power applications, the C5000 ultra low power series is of interest. However, only the TMS320C5509 with a clock frequency of 144 MHz supports a high enough frequency. It can be used for the first partitioning. For all other partitionings a higher clock frequency is required. Another promising TI DSP series is the C67 floating point series. The four devices TMS320C6745, TMS320C6747, TMS320C6726 and TMS320C6727 all have the required peripherals and support frequencies in the stipulated frequency ranges. An overview over the used DSPs, the used clock frequency and a typical price for a 1000 piece quantity is given in Tab. B.7 in App. B.3.

To estimate the power consumption for the chosen DSPs, the TI spreadsheets are used. These sheets require the estimation of the CPU utilization. As mentioned in spreadsheet manuals, CPU utilization is not just the temporal utilization. The intensity of an operation has to be included as well. In this case study, the resampling process is classified as an intensive operation with 90 % utilization, while the rest of the operations of the RTP engine are classified with 30 % utilization. The CPU utilization is then calculated by weighting the temporal utilization with the intensity value. Further, the utilization of the used peripherals has to be estimated. This is done by using the theoretically determined data rates. Details about the estimation of the utilization can be found in App. B.3.

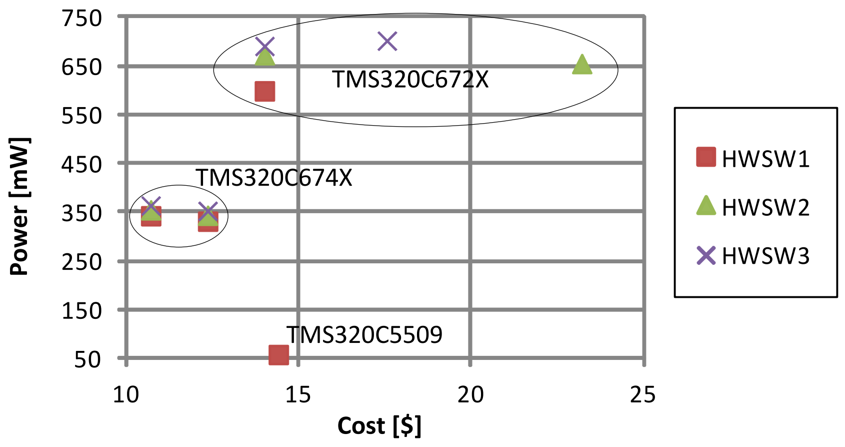


Figure 5.16: Power consumption compared to device costs of the software part. The power consumption significantly depends on the used device. All DSP prices are low compared to the costs of the FPGAs. The underlying data can be found in Tab. B.8.

In Fig. 5.16, the resulting power estimations and the costs for the different solutions are compared. The underlying data of Fig. 5.16 can be found in Tab. B.8. The costs of the different DSPs are varying significantly. However, compared to the costs of the FPGAs, all DSPs are cheap. Concerning the power consumption it can be seen that the processor type has a major impact. The low power DSP TMS320C5509 only consumes 58 mW and therefore is by far the most power efficient solution. For the TMS320C674 family the estimated power consumption is for all partitionings approximately 350 mW. The TMS320C672 series leads to the worst results with an average power consumption of 650 mW. Accordingly, for the first partitioning, the TMS320C5509 is the first choice. For all other partitionings a solution with one of the TMS320C674 DSPs is preferable, since the low power DSP has not enough performance. The difference between the TMS320C6745 and the TMS320C6747 is negligible. The TMS320C6747 is a little bit more expensive, but consumes less power.

Obviously, the power and costs values are just rough estimations. Especially, the estimation of the utilization of a low power DSP by extrapolating the utilization from measurements from the high performance DSP TMS320C6455 introduces inaccuracies. A better estimate can be achieved by using an cycle accurate ISS. However for this case study, this analysis provides sufficiently accurate performance, power and costs estimates.

5.4.3 Costs and Power Comparison

In this Section, the results from the hardware and software analysis are used to estimate the overall power consumption and the overall costs of the different design solutions. Whereby a preselection is made, so that only the hardware and software alternatives with the lowest power consumption are considered. On the software side, this means that only solutions with the low power DSP TMS320C5509 and the TMS320C674 are analyzed.

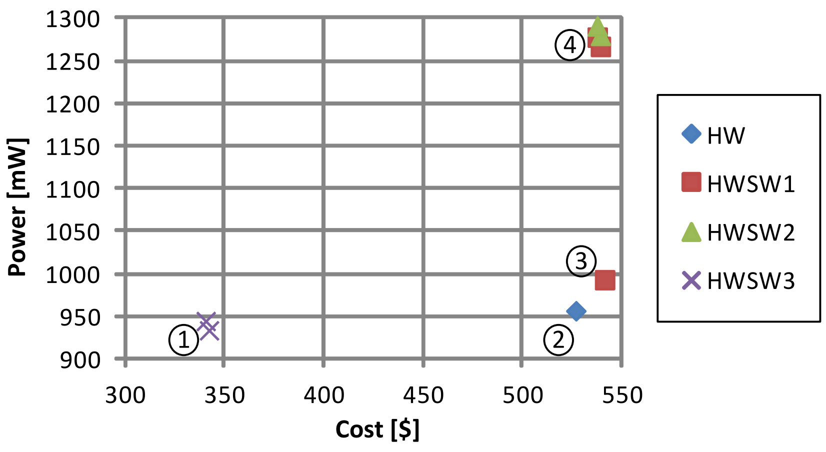


Figure 5.17: Overall power versus costs of different design solutions. The third partitioning HWSW3 results in the most power efficient and cheapest solutions, see ①. A comparable power consumption is achieved by the pure hardware solution, see ②. The underlying data can be found in Tab. B.9.

The cheapest and most power efficient solution is achieved by using the third partitioning HWSW3, see ① in Fig. 5.17. In this case, the low price is achieved since the smaller XC4VFX20 FPGA can be used. Since this FPGA also consumes significantly less power compared to the XC4VFX40, the combination with one of the TMS320C674 processors results in a solution with an estimated power consumption of only 950 mW.

Another power efficient solution is the one chip pure hardware solution, see ② in Fig. 5.17. Since the larger FPGA is used in this case, the price is with $ 526 significantly higher. However, it only consumes a little bit more than 950 mW and since it is a one chip solution, the printed circuit board (PCB) could be designed considerably smaller. Further, the omission of FPGA/DSP communication could reduce the development effort.

More expensive is solution ③ in Fig. 5.17. In this case the first partitioning alternative is used. Since in this case it is possible to use the ultra low power DSP, the power consumption is with 993.5 mW only marginally higher. Less attractive are the partitioning alternatives HWSW1 and HWSW2 with the TMS320C674 DSPs, see ④ in Fig. 5.17. These four solutions have an estimated power consumption of more than 1250 mW and their price is in the range of the expensive solutions.

Following this analysis it becomes apparent that an FPGA/DSP partitioning of the RTP engine only makes sense if this results in a smaller FPGA. Only then it is possible to reduce the overall price. Obviously, not all costs factors have been taken into consideration in this estimate. For a design space exploration of an actual product the projected unit volume is required to estimate the influence of the development costs to the overall costs. However, this analysis gives the designer a feeling for the approximate power consumption and costs and therefore helps to decide the direction into which the product has to be refined.

5.5 Design Effort Analysis

One of the key advantages of the TDA is a reduced design effort for design space explorations. Even if complex data structures are used it is simple to move design components from hardware to software and vice versa. To evaluate this advantage and its impact on the design of a real system, the design effort of the case study is analyzed in the following.

Of course, the analysis of design effort is always difficult. One way design effort can be evaluated is to measure the actual time which has to be spent for developing a system. However, obviously this depends heavily on the designers experience in the used design methodology and in the application area. Therefore, the development time is not an objective metric. Another way of measuring the design effort is the code size. To have a clear definition how this can be measured, the so-called effective lines of code are taken as a metric for design effort. In [53] the effective lines of code (eLOC) are defined as all lines of code which are not comments, blanks or standalone braces or parenthesis. The eLOC are more related to the actual design effort than blank lines or comments, which are often added solely for formatting reasons. However, it is still an approximation. The actual design effort per line of course differs. Some complicated parts of the program require more time to be designed correctly, while other parts like module instantiations in SystemC are relatively simple. Additionally, eLOC values do not consider the effort of several other tasks, which have to be performed for developing a system, e.g. controlling different design tools.

Since effective lines of code are a measurable metric, which provide a general idea of the effort of certain design tasks, they are used in the following to evaluate the presented tripartite design flow. Therefore, the eLOC of the different steps of the performed design space exploration are measured. These values are compared to the estimated lines of code of a traditional design approach. The steps of the design space exploration are the high level modeling, see Sec. 5.5.1, the refinement towards hardware, see Sec. 5.5.2, and the actual design space exploration, which is analyzed in Sec. 5.5.3

5.5.1 High Level Analysis

The following design effort analysis solely focuses on the elements designed using the TDA, hence the RTP engine without RTP filter and jitter buffer, see Sec. 5.2.2. A considerable part of the design process concerns the development of interfaces and the configuration of peripherals. The design effort for all these components is not included in the following analysis. Equally, the top level FPGA files and the main software file are not considered. The design effort analysis

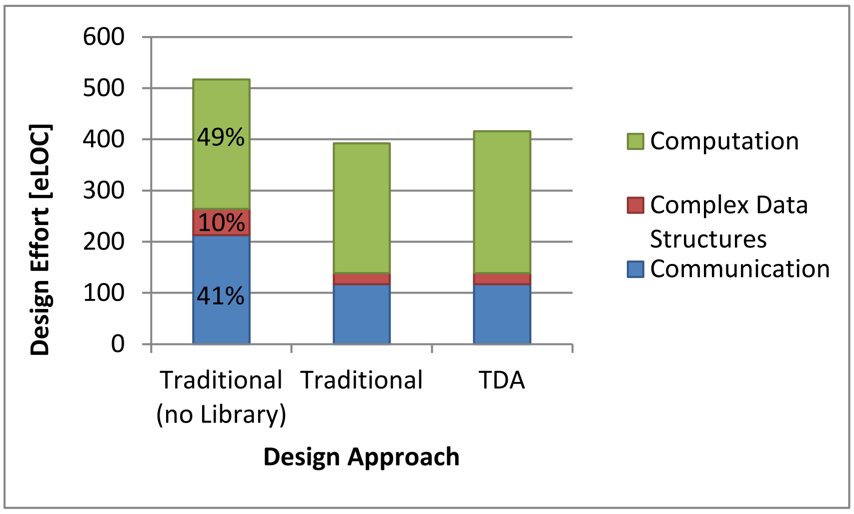


Figure 5.18: The bar labelled "Traditional (no Library)" represents the design effort distribution of the high level RTP engine among communication, computation and data structures. All used lines of code have been taken into account, even function bodies of used library functions. The other two bars represent the actual design effort of a traditional and the Tripartite approach if communication and data structure libraries are used.

solely focuses on the HWSW-Modules and connected communication components. First, the development of the high level model is analyzed.

Counting the lines of code of all involved modules and library components provides a general distribution of the design among communication, computation and data structures, shown in Fig. 5.18 as "Traditional (no Library)". For this analysis, communication and timing channels, e.g. the time difference or the resampler FIFO are classified as communication components. The computation part basically corresponds to the HWSW-Modules. Only the Deque and the Queue, used in the RTP encode unit and in the resampler, are considered as complex data structures. Const Arrays are in this analysis not classified as complex data structures, since they would be realized using simple C/C++ arrays if a traditional design approach was used. Obviously, not all lines of code of used library elements are considered. Only the lines of code of function bodies of actually called library functions are counted. The measured eLOC values of all utilized elements are listed in Tab. B.10, B.11, B.12 and B.13. The resulting distribution of effective lines of code shows that only 10 % of the eLOC is related to complex data structures. The rest is almost uniformly distributed among communication and computation. This result shows that the RTP engine is not a pure data flow system, in which the largest part of the design effort would correspond to computation.

The other two bars in Fig. 5.18 represent the actual design effort of a traditional and the TDA. Whereby, "actual design effort" denotes the effort for a designer to implement the high level model without considering the implementation of the library elements. It is expected that the STL [56] is used as data structure library for the traditional approach. Since the concept of using a communication library is not new, it is also expected that both approaches utilize such a library. The RTP engine additionally has several application specific communication components like the Local Link interface, phase difference and time difference unit and the resampler FIFO. These components have to be designed from scratch independent of the used design approach. As can be seen in Fig. 5.18, the Tripartite Design Approach requires slightly more design effort than a traditional approach. This concerns the computation part, while the effort for communication and data structures is equal. The increased design effort results from the additional keywords in the HWSW-Module, see Sec. 4.1.3.

5.5.2 Refinement to Hardware

The next step of the DSE is the refinement towards hardware. If the TDA approach is used, the HWSW-Modules can be used without any changes. They have to be instantiated in a top level module, which connects them to synthesizable communication adapters and synthesizable CTL container implementations. Details thereto have been presented in Sec. 4.1.3.1. The main design effort concerns the implementation of these top level modules. Measured eLOC values are listed in Tab. B.15. Most communication adapters can be taken from a library. Only the mentioned application specific communication components have to be refined manually. The design effort for refining these components is shown in Tab. B.17. However, this step has to be performed independent of the used approach.

If a traditional design approach is used, many design steps have to be performed manually. First, the STL containers cannot be used. They have to be replaced by synthesizable data structures. To estimate the design effort for this process, the synthesizable CTL container implementations are taken as reference. Again, only the function body of the called library methods are included. Then, data types have to be refined. Their bit width is reduced to save hardware resources. In this case study, the bit width is predefined at many places by VoIP standards. Only a few variables in the RTP encode unit and in the resampler can be refined. The TDA utilizes template parameters to simplify this process. Using a traditional design approach, the declaration of each variable which should be refined has to be modified. Finally, the module structure has to be adapted to the requirements of the used HLS tool. The process type has to be changed to SC_CTHREAD and clock and reset inputs have to be added. Additionally, cycle accurate I/O protocols have to be implemented. To estimate the lines of code of these protocols, the communication adapters of the tripartite approach are considered. The estimated design effort in terms of effective lines of code of all these refinement steps can be found in Tab. B.16.

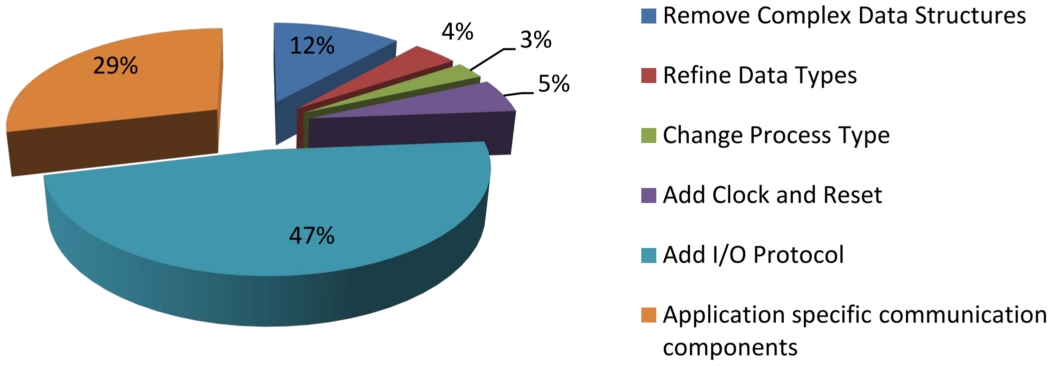


Figure 5.19: Design effort distribution of the hardware refinement step using a traditional design approach.

Fig. 5.19 shows the design effort distribution of all hardware refinement steps which have to be performed if a traditional design approach is used. By far the most time consuming tasks are refinement steps related to communication components. Adding cycle accurate I/O protocols corresponds to 47 % of the overall design effort, while the refinement of application specific communication components corresponds to 29 %. Refinement steps related to complex data structures require comparatively less design effort. Only 12 % of the overall effort correspond to the replacement of STL containers with synthesizable data structure implementations. A reason therefore is the general distribution of the RTP engine among communication, computation and data structures. As presented in Sec. 5.5.1, complex data structures only form 10 % of the whole design.

Comparing the design effort using the TDA and a traditional approach results in an effort reduction of 22 % for the hardware refinement step. If the high level design steps are incorporated as well, still a 9 % reduction can be determined. Additionally, it must be noted that this analysis does not consider the complexity of a line of code. Refining the implementation of the computation module itself, as performed in a traditional design approach, is an error prone process. Whereas, if the TDA is used, predominantely module instantiations and port mappings have to be performed for hardware refinement.

5.5.3 Actual Design Space Exploration

After hardware refinement, different hardware/software solutions are generated and evaluated during the actual design space exploration. Also for a pure hardware design different alternatives can be realized. It is possible to experiment with different design constraints of the HLS tool and the CTL provides the possibility to easily explore different data structure to memory structure mappings.

The used design approach does not affect the effort of applying different design constraints. Even, the design effort of changing the memory mapping of one data structure is independent of the design approach. Only a few lines of code have to be modified to change the realization style from BRAM to registers. An advantage of the CTL elements is the possibility to map different data structures to one and the same memory structure. However, since all data structures of the RTP engine have different bit width, it is not possible to apply this kind of mapping. Actually, only some data structures have a length, where a mapping to BRAM as well as to registers makes sense. Only those design alternatives have been generated during the DSE. The corresponding design effort is listed in Tab. B.18. As expected, the design effort for generating these mappings is very small.

The main part of the design space exploration is the generation of three different hardware/-software partitionings as shown in Fig. 5.8(c). Therefore, the resampler, the G711 encoder and Decoder modules and the RTP encode unit are shifted in three consecutive steps to software. If the Tripartite Design Approach is used, the HWSW-Modules can be used unchanged. Software specific library elements are provided for most communication and data structure elements. A large of the design effort results from the adaption of the application specific communication elements for the used operating system. The corresponding design effort for the three partitionings is shown in Tab. B.19, B.20, B.21. Additionally, the HWSW-Modules have to be connected to the surrounding channels and data structure implementations. However, the design effort of these port mappings is comparatively negligible.

Several other design steps have to be performed if a traditional design approach is used. Fig. 5.20 shows a distribution of the overall design effort of the three design steps among all required tasks. Since the computation modules have been changed significantly for hardware design, it is reasonable to start again with the high level model. Thus complex data structures have to be removed again. In principle, the utilization of the STL is possible, since software ports exist for many modern DSPs. However, the case study has shown that dynamic memory management significantly increases the execution time, see 5.4.2. Again, data types have to be refined and OS specific synchronization facilities have to be added. Finally, all SystemC constructs have to be removed to transform the computation modules to a pure C++ classes. The design effort of these refinements is listed in Tab. B.19, B.20, B.21

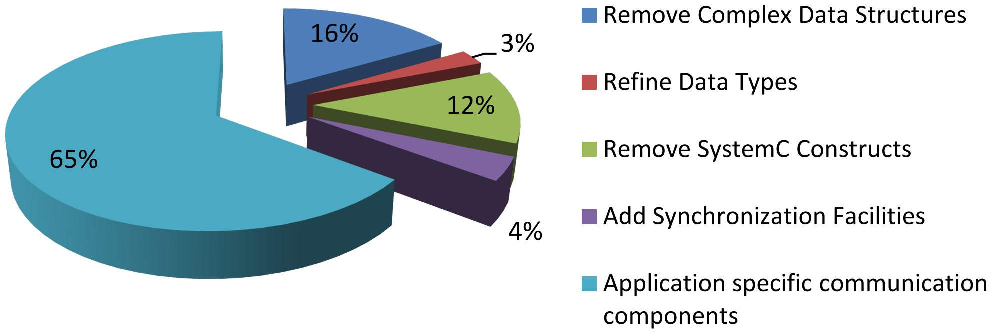


Figure 5.20: Design effort distribution of moving elements from hardware to software using a traditional design approach.

Fig. 5.20 shows that the effort of porting application specific communication components corresponds to approximately two thirds of the overall design effort. This is the part which has to be performed anyway independent of the applied design approach. However, removing complex data structures and SystemC constructs corresponds to almost one third of the overall design effort. Both steps can be completely skipped if the TDA is used. This results in a reduction of the design effort of 27 % for shifting the mentioned modules to software.

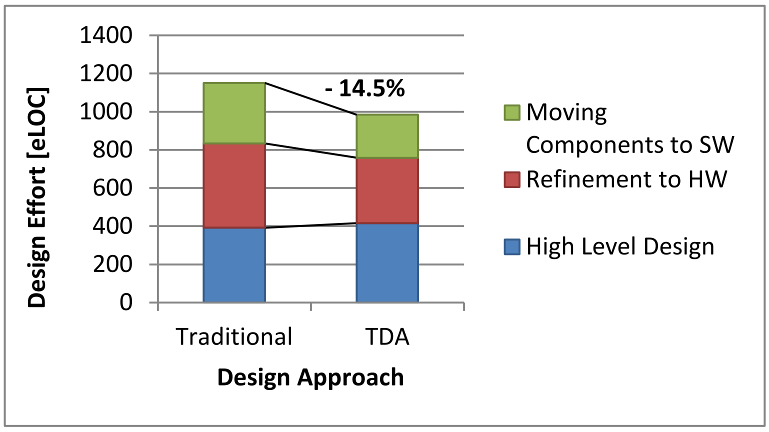


Figure 5.21: Overall design effort comparison of a traditional versus the Tripartite Design Approach. The overhead for implementing realization independent HWSW-Modules for high level design is compensated by the savings during the design space exploration. This results in a total reduction of 14.5 % of the design effort.

A comparison between the Tripartite and a traditional approach of the overall effort is listed in Tab. B.22 and illustrated in Fig. 5.21. Many tasks have to be performed anyway independent of the applied design approach and additional design effort is required at the system level. However, the significant effort reduction of 22 % for the refinement towards hardware and 27 % for the generation of different HW/SW partitionings result in a total reduction of 14.5 % for the whole design space exploration.

5.6 Summary and Evaluation

The aim of this case study was the application of the Tripartite Design Approach and the CTL library to a real world example. Therefore, an embedded VoIP engine optimized for safety critical application areas has been chosen. Several design constraints have to be fulfilled. The expected small to medium production quantity forces the use of components off-the-shelf like FPGAs and general or special purpose processors. The safety critical application leads to further specific design requirements like a minimum end-to-end delay. The fulfillment of these requirements significantly influenced the architecture design. A small jitter buffer and the utilization of a clock recovery combined with a resampler, helped to achieve a very short delay. The application of this resampler transformed the originally data flow dominated design to a heterogeneous system with two clock domains. Such systems cannot be handled by many ESL solutions, which solely focus on pure data flow dominated designs. Four different prototypes have been realized using FPGA and DSP evaluation boards. Several tests and measurements confirmed their principle functionality and proofed the fulfillment of a small end-to-end delay.

A part of the architecture was already fixed by a previous work, see [Wen11]. Part of the VoIP stack has been realized on a PPC on an FPGA. The integrated MAC core has been utilized for an Ethernet interface. One open question remained. The RTP engine could be realized solely on the FPGA or parts of it could be moved to a DSP. To find the solution, which fits best to the given design constraints, minimum power consumption and minimum component costs, a design space exploration has been performed. The results show that the partitioning of the design to an FPGA and a DSP can reduce the overall price and power consumption. Moving four modules to software results in an FPGA design, which fits on a smaller Virtex4 device. The resulting HW/SW design has the lowest component costs and the lowest power consumption.

Finally, the design effort of the design space exploration has been analyzed. It has been shown that the advantages of the TDA and the CTL library actually lead to a reduced design effort. Although it is difficult to measure such design effort reduction, the performed analysis, which is based on eLOC values, provides a general idea of the required design effort. The comparison to a traditional approach showed a significant effort reduction of almost 15 % for the whole design space exploration. A part of this reduction is based on the CTL library, which allows the usage of complex data structures for design space exploration. Another simplification originates from the concept of the realization independent computation module. The presented facilities signifcantly simplify transforming a high level computation component to either a synthesizable hardware module or to a pure C++ class.

6 Discussion

In this work the Tripartite Design Approach for the design space exploration of board-level systems has been presented. Four different requirements for an ESL solution specifically for board-level systems have been identified in Sec. 1.1.4. Now, the fulfillment of these requirements is analyzed. Finally, the achievement of the most important goal, the reduction of the overall design effort is discussed. The analysis is based on the perceptions and results gained in the case study, where the design space exploration of a VoIP engine has been been performed.

High Level of Abstraction

The application-to-architecture mapping is performed by using a high level functional model. This first functional model of a system typically consists of algorithms implemented using high level language features. Therefore, the support of high level language constructs and a high abstraction level in general have been identified as an important feature of an ESL environment. In general, the Tripartite Design Approach supports all language features, which are accepted by modern HLS tools. This includes features such as object oriented design and generic programming via template parameters.

The analysis of existing solutions and capabilities of HLS tools has shown that all of them have limitations regarding complex data structures. The reasons therefore are mainly difficulties with dynamic memory management and heavy use of pointers. The approach presented in this work solves this by separating data structures from computation by utilizing the abstract data type concept. As shown in the case study this truly increases the abstraction level by enabling the use of complex data structures like deque, queue or list. The possibility to exchange the data structure implementation not only allows the removal of non-synthesizable constructs for hardware design, it also allows the optimization for the respective target.

However, the adaption of the iterator concept for the CTL List container showed the limitations of this approach. The important step to enable complex data structures for design space exploration is the encapsulation of implementation details. Thereby, unsupported language constructs can be avoided in the synthesized computation. Though, with STL facilities like iterators or functors, this encapsulation is not possible anymore. The realization of such software concepts requires the utilization of pointers in the computation part. Hence, whether the abstraction level can be further increased or not, heavily depends on the features of the utilized HLS tool.

Link to Implementation

Another requirement for an ESL solution is the link to the implementation. This is especially important for low volume embedded systems, since prototyping and in system debugging plays a major role for their development. On the one hand board level systems with FPGAs and special or general purpose processors can be modified easily after production, which is different for SoC designs. On the other hand the importance of prototyping results from a lack of system simulation capabilities. Furthermore, if the high level model can easily be translated to an actual hardware/software implementation, this allows the application of low level tools for accurate design parameter estimation.

In the case study, all four basic HW/SW partitionings have been realized as prototype using FPGA and DSP evaluation platforms. This shows the simplicity and rapidity of the prototype generation with the TDA. Obviously, a certain effort has to be made to setup a first HW/SW prototype. Considering the VoIP engine of the case study, e.g. an initial startup for both boards was required. Furthermore, a connection between the FPGA and the DSP board had to be designed. However, after the first prototype was finished, the realization independent design of computation modules ensured the simple generation of several further prototypes. Only a few application specific communication components had to be refined manually.

The realization independence of computation is realized using the so-called HWSW-Module. This module can be transformed into a synthesizable or compilable module via preprocessor macros and template parameters. Thereby, the integration of an HLS tool plays a key role. Since HLS allows the translation of untimed computation code to a hardware implementation, computation can be modeled synthesizable and compilable. However, the realization independence has its limits. Already the design of the algorithm influences the final hardware resource requirements. It is often possible to precalculate certain parts of the algorithm. While this reduces the calculation effort during runtime, it increases the storage requirements. It might be useful in a software realization to precalculate as much as possible, since there is enough memory. However, in an FPGA the opposite solution might be preferable. This shows that it still might be necessary to modify a realization independent HWSW-Module to explore different design solutions.

Full Target Support

The third of the four requirements, which have been identified in Sec. 1.1.4, is the full target support. It is not sufficient to find any mapping from the high level model to the desired platform. The generated implementation is used to estimate design parameters via low level tools. These parameters are the basis for an evaluation of a high level design decision like the HW/SW partitioning. The more efficient the implementation, the more accurate are the design parameter estimates and the more accurate is the evaluation. Existing solutions focus mainly on SoC design. This results in insufficient support of FPGAs. For FPGA design the efficient utilization of the available resources is important. This concerns for example memory structures. Modern HLS tools require the manual utilization of memory structures. Therefore, an ESL environment, which integrates an HLS tool, has to find the best data structure to memory structure mapping automatically or it provides a mapping possibility for the designer.

The CTL leaves the mapping decision to the user. The concept of separating the complex data structure from the actual memory structure enables the variation of the mapping without modifying the data structure implementation itself. Thereby, different mapping solutions can be explored without much design effort. The possibility to map several data structures to one and the same memory structure further ensures an efficient utilization of available RAM structures. However, the current implementation has certain limitations, e.g. data structures can only be mapped to the same memory structure if they have the same bit width. To improve the applicability, the existing limitations have to be removed and further memory structures such as external memory chips have to be added. Anyhow, the presented mapping feature works as proof of concept and in many cases, it enables the efficient mapping of complex data structures to FPGAs.

However, the efficient utilization of memory structures is not sufficient to ensure the best application to target mapping. Modern FPGAs and SPPs usually have certain special components which have to be supported to optimally exploit the target hardware. SPPs for example have DMAs or special coprocessors which reduce the workload of the CPU. Also FPGAs often provide several special purpose cores, like high speed interfaces, which increase the implementation's performance. The generation of accurate design parameter estimates can be improved if these components are supported by the design flow.

Another issue, which is related to the target support, is the task execution structure on the SPP. Currently all computation threads are executed in parallel. In larger designs, this may lead to an overhead due to the time spent for context switches. Often, a dynamic scheduling of all tasks is not necessary. In this case, it is possible to schedule the threads statically. All these optimizations increase the performance of the generated HW/SW prototypes. Although, it is not required to reach the optimum, more optimized prototypes allow a more accurate evaluation of different HW/SW partitionings.

Application Domain Independence

Another identified requirement is the application domain independence. Although the main focus of this work is on data flow dominated designs, the developed ESL solution has to support control flow oriented parts as well. Some of the existing ESL tools are restricted to pure data flow designs. This simplifies tool development, but it reduces the applicability of the ESL solution. For modeling data flow oriented designs, a pure data flow MoC can often only be utilized at the highest level of abstraction. After refinement towards the actual implementation timing and events become important. These kind of systems cannot be designed using an ESL solution, which is restricted to pure data flow applications.

Comparable to process based MoCs, the TDA basically consists of untimed processing nodes implemented via the HWSW-Module. However, the introduction of timing channels enables the integration of timing and events into the system model. Thereby, as shown with the case study, also applications, which are not pure data flow systems, can be handled. At a high abstraction level, the RTP engine is an audio processing unit, which consists of pure data flow components. However, frequency inaccuracies as they appear in reality required the addition of an audio resampler module which performs a conversion between two clock domains. To accurately model this system, timing and events are needed. In the case study, the presented concept of the timing channels has been utilized to correctly model the system.

With the introduction of timing channels, the presented Tripartite Design Approach has the same expressiveness as the discrete event MoC. Basically all systems which can be modeled in SystemC, can be modeled via the TDA. Thus, the aim of application domain independence is achieved. A disadvantage of this concept is, that a certain amount of complexity is moved to the channel design.

Design effort

The most important design goal for the TDA is the reduction of the overall design effort. The developed design flow has to reduce the effort for performing a design space exploration. The case study has shown, that the TDA reduced the modeling effort for this particular design by almost 15 %. Although the TDA leads to a small modeling overhead at the system level, the effort reductions for generating different HW/SW partitionings more than compensate this overhead. The concept for transforming a HWSW-Module to a synthesizable hardware module already reduced the design effort by 22 % compared to a traditional approach. A major part of the reduction is achieved by the utilization of the adapter concept [GLMS02, p. 156]. Instead of manually adding I/O protocols into the computation module, protocol adapters from the library are connected to the HWSW-Module.

After the refinement of the high level model to a first pure hardware solution, three HW/SW partitionings have been generated by mapping different HWSW-Modules to a DSP. The design effort reduction for this step amounts to 27 %. The advantage of the TDA thereby has basically two reasons. One reason is the realiziation independent HWSW-Module, which can be used for hardware and software without modifying the source code. In a traditional approach, the high level SystemC module has to be transformed manually to a pure C++ class. The second reason is the support of complex data structures. If a traditional design approach is used, the complex data structures have to be replaced manually. Additionally, different implementations for hardware and software are necessary. If the TDA and the CTL are utilized, complex data structures can be used at the system level and their implementation can easily be replaced with optimized hardware or software specific implementations.

Therefore, the TDA significantly reduces the modeling effort for design space explorations, where different HW/SW partitionings have to be generated. In the case study only 10 % of the design were related to complex data structures. The more complex data structures are used, the more the designer benefits from using the TDA.

The design effort analysis in the case study primarily concentrated on the modeling effort in terms of effective lines of code. However, for the generation of various different HW/SW partitionings and for the estimation of design parameters such as power, costs and performance, several tools have to be executed in different design steps. To analyze the power consumption of an FPGA design, first a high level synthesis tool is applied. Then, logic synthesis and place and route are applied to the generated RTL code. To estimate the switching activity of a design, a post layout simulation can be performed. The final netlist and the switching activity are then used by the power analyzer tool to calculate the estimated power consumption of different components of the design. Hence, several tools and design steps have to be performed to get an accurate estimate.

The automation of at least a part of these steps would further decrease the design effort. Obviously, a fully automated ESL environment even with an integrated evaluation of the different design solutions would result in the least effort for the designer. However, each automation typically leads to less flexibility. The case study showed, that a library based approach has its advantages. The design is automated as long as standard components from the library can be used. Some application specific components related to the resampler have been designed and refined manually. Therefore, a library based approach can always be extended and modified for application specific needs. In a fully automated environment this is typically not possible.

7 Conclusion and Outlook

The aim of this work was the development of an electronic system level environment which specifically supports board-level systems. The analysis of existing solutions has shown that most solutions focus on SoC design. Additionally, all of these solutions have limitations regarding complex data structures. This significantly reduces the level of abstraction and increases the effort of designs, which utilize complex data structures.

The Tripartite Design Approach presented in this work, solves this problem by applying a threefold separation for system level modeling. Not only communication but also data structures are separated from computation. Thereby, complex data structures can be supported at the system level. Their refinement towards hardware or software is handled via a library based approach. The Codesign Template Library provides several commonly used containers and a methodology to easily replace the implementation of the container. The main advantage of the TDA is the increased abstraction level by supporting complex data structures. Simultaneously, the CTL provides a solid link to optimized hardware and software implementations. Besides this, the concept of realization independent computation modeled via HWSW-Modules, simplifies shifting components from hardware to software and vice versa. This reduces the design effort for generating various different HW/SW partitionings. The case study has shown that even the generation of different prototypes is simplified in this way. A link to the implementation and a fast prototype generation have been identified as important features for board-level systems.

The main focus of the TDA is on data flow oriented designs. The modeling approach is comparable to a process based MoC. However, its basic model of computation is SystemC's discrete event MoC. By providing timing channels, timing and events can be included into the system level model. In this way, basically all systems which can be designed with SystemC can be modeled via the TDA. The advantage can be seen in the case study. The VoIP engine is a data flow dominated design, which also includes control flow components. This system could not be designed with an ESL solution restricted to pure data flow designs.

Since most ESL tools focus mainly on SoC designs another goal of this work was to provide a solution specifically supporting board-level systems utilizing FPGAs and dedicated processors. To fulfill the particular requirements of these targets, a memory mapping facility has been added to the CTL. This feature enables the efficient mapping of data structures to memory structures on the FPGA. This is an important step to efficiently utilize the resources available on the target architecture. However, the analysis has shown that several additional optimizations can be added to increase the target support. On the one hand the support of special components like DMAs, hardware timers or specific interface cores can be added in the future. On the other hand, the software execution structure can be optimized to increase the performance of the generated software implementations. More optimized implementations lead to better prototypes and more accurate design parameter estimates. Anyhow, with the current solution many applications can be mapped efficiently to FPGA/SPP architectures. Additional optimizations can be added in the future to increase the applicability and to optimize the generated results.

Besides that, possible future work concerns the extension of the component libraries and the automation of parts of the design flow. The communication library currently only consists of components needed for the case study. Several different communication and timing channels can be added to increase the design capabilities. A certain impact can be expected by providing generic implementations for several common hardware/software interfaces. Obviously, also the extension of the CTL with further containers, different implementation options and additional mapping possibilities would increase the value of this library. Another possibility for future work concerns the automation of parts of the design flow. Thereby, it is important that the flexibility due to the library based approach is not given up. However, a semi-automated design flow would further decrease the effort required to perform an exhaustive design space exploration.

A Codesign Template Library

In this Chapter implementation details of the CTL are presented. First, Sec. A.1 shows the basic class declarations of all seven container adapters. These class declarations give an overview over the various interface functions provided by the CTL elements. Then, Sec. A.2 presents particular code snippets, which help to better understand the implementation principle of the CTL.

A.1 Interfaces of CTL Containers

In this Section the basic class declaration of the container adapters of all seven CTL elements is presented. This basic declaration includes function prototypes, type definitions and member variables.

Listing A.1: Array

```
template <typename Container>
class const_array
{
public:
  typedef typename Container::size_type size_type;
  typedef typename Container::data_type T;

  Container mContImpl;

  //Element Access
  T read(size_type n);
};
```

Listing A.2: Const Array

```
template <typename Container>
class array
{
public:
  typedef typename Container::size_type size_type;
  typedef typename Container::data_type T;

  Container mContImpl;

  //Element Access
  T read(size_type n);
  void write(T u, size_type n);
};
```

Listing A.3: Vector

```
template <typename Container>
class vector
{
public:
  typedef typename Container::size_type size_type;
  typedef typename Container::data_type T;

  Container mContImpl;

  template <typename Cont>
  vector<Container>& operator= (vector<Cont>& x);

  //Capacity
  size_type size() const;
  size_type max_size() const;
  void resize(size_type sz, T c = T());
  size_type capacity() const;
  bool empty() const;
  void reserve(size_type n);

  //Element Access
  T read(size_type n);
  void write(T u, size_type n);
  T front();
  T back();

  template <typename Cont>
  void assign(Cont& _cont, size_type _addr, size_type _size);
  void assign(size_type n, const T u);
  void push_back(const T x);
  void pop_back();

  void insert(size_type position, T x);
  void insert(size_type position, size_type n, T x);
  template <typename Cont>
  void insert(size_type position, Cont & _cont, size_type _addr,
    size_type _size);

  void erase(size_type position);
  void erase(size_type _addr, size_type _size);

  template <typename Cont>
  void swap(vector<Cont>& vec);
  void clear();
};
```

Listing A.4: Deque

```
template <typename Container>
class deque
{
public:
  typedef typename Container::size_type size_type;
  typedef typename Container::data_type T;

  Container mContImpl;

  template <typename Cont>
  deque<Container>& operator= (deque<Cont>& x);

  //Capacity
  size_type size() const;
  size_type max_size() const;
  void resize(size_type sz, T c = T());
  bool empty() const;

  //Element Access
  T read(size_type n);
  void write(T u, size_type n);
  T front();
  T back();

  //Modifiers
  template <typename Cont>
  void assign(Cont& _cont, size_type _addr, size_type _size);
  void assign(size_type n, T u);

  void push_back(T x);
  void push_front(T x);
  void pop_back();
  void pop_front();

  void insert(size_type position, T x);
  void insert(size_type position, size_type n, T x);
  template <typename Cont>
  void insert(size_type position, Cont & _cont, size_type _addr,
    size_type _size);

  void erase(size_type position);
  void erase(size_type _addr, size_type _size);

  template <typename Cont>
  void swap(deque<Cont>& deq);
  void clear();
};
```

Listing A.5: Stack

```
template <typename Container>
class stack
{
```

```
public:
  typedef typename Container::size_type size_type;
  typedef typename Container::data_type T;

  Container mContImpl;

  //Capacity
  size_type size() const;
  bool empty() const;

  //Element Access
  T top();

  //Modifiers
  void push(const T x);
  void pop();
  void clear();
};
```

Listing A.6: Queue

```
template <typename Container>
class queue
{
public:
  typedef typename Container::size_type size_type;
  typedef typename Container::data_type T;

  Container mContImpl;

  //Capacity
  size_type size() const;
  bool empty() const;

  //Element Access
  T front();
  T back();

  //Modifiers
  void push(T x);
  void pop();
  void clear();
};
```

Listing A.7: List

```
template <typename Container>
class list
{
public:
  typedef typename Container::size_type size_type;
  typedef typename Container::data_type T;

  Container mContImpl;

  class iterator
  {
  public:

    typedef typename Container::iterator ContIter;

    iterator(const ContIter & cit);
    iterator(const iterator & it);
    iterator& operator= (const iterator & it);
    bool operator==(const iterator & it);
    bool operator!=(const iterator & it);

    ContIter mIterImpl;
  };

  template <typename Cont>
  list<Container>& operator= (list<Cont>& x);

  //Iterators
  iterator begin();
  iterator end();
  void increment(iterator& iter, size_type n = 1);
  void decrement(iterator& iter, size_type n = 1);
  T read(iterator iter);
  void write(T u, iterator iter);

  //Capacity
  size_type size();
  size_type max_size();
  void resize(size_type sz, T c = T());
  bool empty();

  //Element Access
  T front();
  T back();

  //Modifiers
  template <typename TCont, typename TIter>
  void assign(TCont& _cont, TIter& first, TIter& last);
  void assign(size_type n, T u);

  void push_back(T x);
  void push_front(T x);
  void pop_back();
  void pop_front();

  void insert(iterator iter, T x);
  void insert(iterator iter, size_type n, T x);

  template <typename TCont, typename TIter>
  void insert(iterator iter, TCont& _cont, TIter first, TIter last);
  void erase(iterator iter);
  void erase(iterator first, iterator last);

  template <typename Cont>
  void swap(list<Cont>& lis);
  void clear();

  //Operations
  void remove(const T value);
  void unique();
  void reverse();
};
```

A.2 Code Examples of CTL Containers

In this Section three different code examples of CTL container implementations are shown. In Lst. A.8 a code snippet of the high level implementation of the Vector is illustrated. The implementation of the `push_back()` function is shown as an example for the various functions the Vector provides. Like in all high level implementations, a standard STL container is used to store the elements, see line 15. As explained in Sec. 4.2.2.2, each container implementation has to define two data types called `data_type` and `size_type`. The data type used for addresses (`size_type`) is in this case derived from the used STL container (line 5 in Lst. A.8). The second type (`data_type`), is the data type of the stored elements. It is set via the template parameter `T` (line 6 in Lst. A.8). The variable `mMaxSize` logs the maximum size of the container during a simulation run.

Listing A.8: Code snippet of the Vector's high level implementation.

```
 1 template <typename T>
 2 class hl_vector
 3 {
 4 public:
 5   typedef typename std::vector<T>::size_type size_type;
 6   typedef T data_type;
 7   ...
 8   void push_back(data_type x)
 9   {
10     mData.push_back(x);
11     if (this->mMaxSize < mData.size())
12       this->mMaxSize = mData.size();
13   }
14
15   std::vector<data_type> mData;
16   size_type mMaxSize;
17 };
```

The second Listing shows the hardware implementation of the same function, see Lst. A.9. It has three template parameters (line 1). The additional two parameters are the bit width of the address data type (`AddressSize`) and the static size of the container (`__Size`). In line 6 a special keyword, which is required for the HLS tool, is added. Since it is not needed for simulation, preprocessor directives are utilized so that it is only compiled during synthesis runs. The implementation of the `push_back()` function operates on the connected memory structure. The actual connection is realized via an `sc_port`, see line 23. Another particularity is the offset variable defined in line 21. It is used to map the data structure to a certain address range of the memory structure. By mapping several data structures to different address ranges of one and the same memory structure, an efficient memory structure utilization is ensured.

Listing A.9: Code snippet of the Vector's hardware implementation.

```
 1 template <typename T, int AddressSize, int __Size>
 2 class hw_vector
 3 {
 4 public:
 5   #ifdef CYNTHESIZER
 6     CYN_INLINE_MODULE;
 7   #endif
 8
 9   ...
10   void push_back(T x)
11   {
12     assert(int(mSize) < __Size);
13     if (int(mSize) < __Size)
14     {
15       mData->write(x, mOffset + int(mSize));
16       ++mSize;
17     }
18   }
19   ...
20
21   int mOffset;
22   size_type mSize;
23   sc_port<ram_port_if<T> > mData;
24 };
```

The third code snippet shown in Lst. A.10, presents two iterator related functions of the List's hardware implementation. The first of these functions, `read()`, corresponds to dereferencing the iterator, see line 1. Since no real C++ pointers are used, this function has to be implemented in the List implementation. The address stored in the iterator is used to read the element, to which the iterator points. The second function, `increment()`, modifies an iterator, so that it points to the next list element. The iterator is passed per reference as function parameter, see line 9. Then, it is dereferenced. Hence, the element to which it points is read. This element stores the address of the following list element (`v.pNext` in line 21). Finally, the address stored in the iterator is updated.

Listing A.10: Hardware implementation of the iterator functions `increment()` and `read()`.

```
1 data_type read(iterator iter)
2 {
3   assert(!iter.mEnd);
4
5   list_type v = to_list_type(mData->read(mOffset + int(iter.mAddress)));
6   return v.Value;
7 }
8
9 void increment(iterator& iter, size_type n = 1)
```

```
{
  for (int i=0; i<int(n); i++)
  {
    if (iter.mAddress == mLast)
      iter.mEnd = true;
    else
    {
      //read list element
      list_type v =
        to_list_type(mData->read(mOffset + int(iter.mAddress)));
      //increment iterator
      iter.mAddress = v.pNext;
    }
  }
}
```

B Case Study

In this Chapter different further details concerning the case study are shown. This includes further measurement results, details regarding the cost and power estimation of the different HW/SW partitionings and details regarding the design effort analysis.

B.1 Further Measurements

In the following frequency spectra of different other measurements are shown. The simulation result compared to the measurements of one of the HW/SW prototypes is shown in Fig. B.1. In Fig. B.2 the pure hardware prototype is directly compared to one of the HW/SW prototypes.

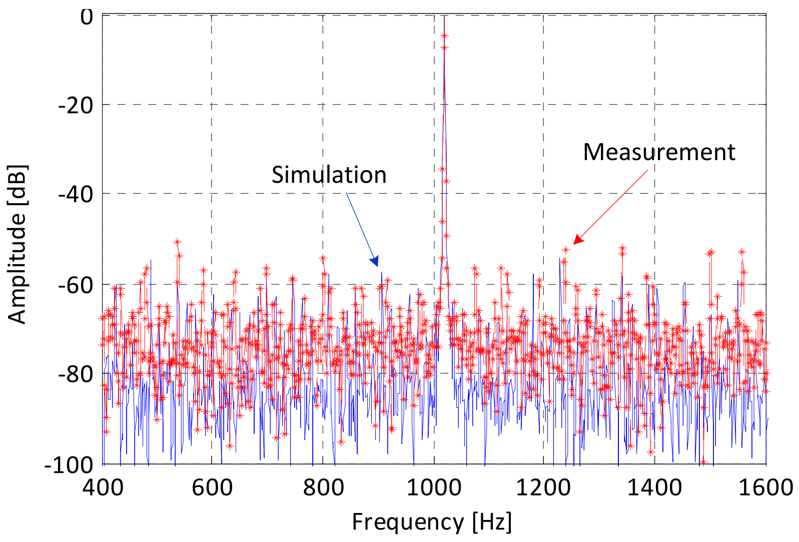


Figure B.1: Simulation versus measurement of one of the HW/SW prototypes.

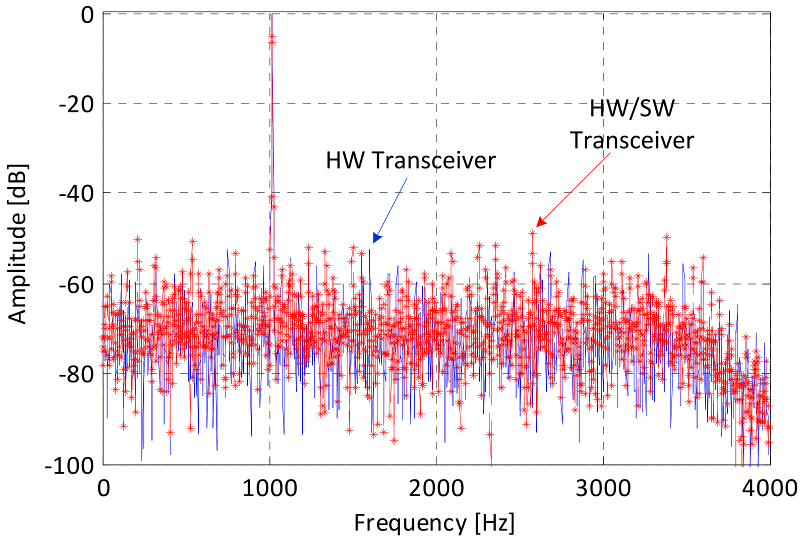


Figure B.2: Direct comparison of the pure HW prototype with one of the HW/SW prototypes.

B.2 Hardware Resource and Power Analysis

In the Section, detailed resource and power information about the hardware design components are given.

Table B.1: Hardware resources of all mentioned design components including the different data structure mappings are shown. RAM and ROM stand for mapping to registers, while BRAM and BROM correspond to a mapping to a block RAM.

	Registers	Slices	LUTs	BRAMs
PowerPC	6034	5235	6561	33
Ethernet MAC	445	331	478	4
RTP Filter	396	338	539	0
Jitter Buffer	88	159	305	1
Jitter	437	738	808	0
Clock Recovery	129	271	511	0
G711 Encode	157	118	196	4
HWSW1 EMIF	47	44	72	2
HWSW2 EMIF	54	47	78	2
HWSW3 EMIF	105	114	177	2
G711 Decode BROM	79	55	66	1
G711 Decode ROM	61	61	124	0
Resampler RAM/ROM	2118	2726	4993	0
Resampler RAM/BROM	2189	2217	4000	1
Resampler BRAM/BROM	772	707	1313	2
RTP Encode RAM/ROM	1584	2202	4138	0
RTP Encode RAM/BROM	1592	2262	4240	1
RTP Encode BRAM/ROM	672	1500	2844	1
RTP Encode BRAM/BROM	715	1447	2753	2

Table B.2: Power and resource estimation for different partitionings and for each partitioning different memory structure mappings are compared.

Parititioning	Memory Structure			Power	Slices	BRAMs
	Decode G711	Resampler	RTP Encode			
HW	ROM	RAM	RAM/ROM	957 mW	7866	10
	ROM	BRAM	RAM/ROM	1046 mW	6356	11
	BROM	BRAM	RAM/ROM	1059 mW	6350	12
	BROM	BRAM	RAM/BROM	1057 mW	6410	13
	BROM	BRAM	BRAM/BROM	1027 mW	5595	14
	BROM	BRAM	BRAM/ROM	1030 mW	5632	13
HWSW1	ROM	-	RAM/ROM	1015 mW	4262	11
	BROM	-	RAM/ROM	989 mW	4256	12
	BROM	-	RAM/BROM	996 mW	4316	13
	BROM	-	BRAM/BROM	935 mW	3501	14
	BROM	-	BRAM/ROM	942 mW	3538	13
HWSW2	-	-	RAM/ROM	969 mW	4086	7
	-	-	RAM/BROM	981 mW	4146	8
	-	-	BRAM/BROM	936 mW	3331	9
	-	-	BRAM/ROM	952 mW	3368	8
HWSW3	-	-	-	581 mW	1951	7

B.3 DSP Power Estimation

Details about the power and cost estimation of the DSP implementations are presented in this Section. Tab. B.3 shows the time measurements of different tasks with a clock frequency of 1 GHz. Based on these measurements, runtime estimations for different other clock frequencies have been extrapolated. The shown percentage corresponds to the temporal utilization of the CPU.

Table B.3: Measured (1 GHz) and estimated runtime for different frequencies. Provided percentages show the relation of the runtime to the 125 ns period.

Solution	Clock Frequ.	Idle		Resampler		Rest	
HWSW1	1000 MHz	111 770 ns	89.42 %	9230 ns	7.39 %	4000 ns	3.2 %
	144 MHz	33 125 ns	26.5 %	64 097 ns	51.28 %	27 777 ns	22.22 %
	225 MHz	59 179 ns	47.34 %	45 920 ns	36.74 %	19 900 ns	15.92 %
	375 MHz	89 720 ns	71.78 %	24 613 ns	19.69 %	10 666 ns	8.53 %
HWSW2	1000 MHz	104950 ns	83.96 %	9230 ns	7.39 %	10 820 ns	8.66 %
	250 MHz	42 149 ns	33.72 %	38 140 ns	30.51 %	44 711 ns	35.77 %
	266 MHz	49 624 ns	39.7 %	34 699 ns	27.76 %	40 677 ns	32.54 %
	375 MHz	71 533 ns	57.23 %	24 613 ns	19.96 %	28 853 ns	23.08 %
HWSW3	1000 MHz	102917 ns	82.33 %	9230 ns	7.39 %	12 853 ns	10.28 %
	266 MHz	34 338 ns	27.47 %	34 699 ns	27.76 %	55 962 ns	44.77 %
	275 MHz	37 305 ns	29.84 %	33 564 ns	26.85 %	54 131 ns	43.30 %
	375 MHz	60 691 ns	48.55 %	24 613 ns	19.69 %	39 696 ns	31.76 %

The Calculated utilizations of the McBSP, of the GPIOs used for clock signal exchange and of the EMIF interface are shown in Tab. B.4 to B.6.

Table B.4: Utilization estimation of the Multichannel Buffered Serial Port (McBSP).

Peripheral	Amount of Data	Write	Frequency	Utilization
McBSP	1 x 32 bit	50 %	20 MHz	1 %

Table B.5: Utilization estimation of General Purpose Input/Output (GPIO) pins.

Peripheral	# of Pins	# of Outputs	Min. Frequency	Utilization
GPIO	2	1	144 MHz	0 %

Table B.6: Utilization estimation of the External Memory Interface (EMIF).

Peripheral	Partitioning	Amount of Data	Write	Min. Frequency	Utilization
EMIF	HWSW1	4 x 16 bit	50 %	144 MHz	0 %
	HWSW2	2 x 16 bit	50 %	144 MHz	0 %
	HWSW3	38 x 16 bit	98 %	144 MHz	1 %

Tab. B.7 shows chosen DSP devices. It shows the operating frequency, DSP price and the CPU utilization for the used RTP Engine partitioning. HWSW1 corresponds to the first partitioning in Fig. 5.8, HWSW2 to the second and HWSW3 to the third. The prices have been taken from [22](30/09/2011) and are for a quantity of 1000 devices. The percentage for a given DSP and HWSW solution combination corresponds to the calculated CPU utilization.

Table B.7: Different DSPs used for different HW/SW partitionings. The Table shows the estimated CPU utilization and DSP prices.

DSP	Clock Frequency	Price	CPU Utilization HWSW1	HWSW2	HWSW3
TMS320C5509A	144 MHz	$ 14.40	52.81 %		
TMS320C6745	375 MHz	$ 10.70	20.28 %	24.65 %	27.25 %
TMS320C6747	375 MHz	$ 12.35	20.28 %	24.65 %	27.25 %
TMS320C6726	225 MHz	$ 14.00	37.84 %		
TMS320C6726B	266 MHz	$ 14.00		34.75 %	38.41 %
TMS320C6727B	250 MHz	$ 23.20		38.19 %	
TMS320C6727B	275 MHz	$ 17.55			37.16 %

The calculated power consumption using the estimated utilization of the peripherals and the CPU is listed in Tab. B.8.

Table B.8: Power estimation and costs for different DSP realizations.

Parititioning	DSP	Estimated Power	Costs
HWSW1	TMS320C5509	58.5 mW	$ 14.4
	TMS320C6745	343 mW	$ 10.7
	TMS320C6747	332 mW	$ 12.35
	TMS320C6726	599 mW	$ 14
HWSW2	TMS320C6747	344 mW	$ 12.35
	TMS320C6745	355 mW	$ 10.7
	TMS320C6727	655 mW	$ 23.2
	TMS320C6726	673 mW	$ 14
HWSW3	TMS320C6747	353 mW	$ 12.35
	TMS320C6745	364 mW	$ 10.7
	TMS320C6727	702 mW	$ 17.55
	TMS320C6726	691 mW	$ 14

B.4 Costs and Power Estimations of Different Design Solutions

This Section presents the details of the costs and power analysis of the complete HW/SW systems. In Tab. B.9 cost and power estimates of different design solutions are shown. For all different alternatives, the exact type of the used FPGAs and DSPs is listed. Power and cost estimates correspond to the values used in Fig. 5.17.

Table B.9: Cost and power estimation for different realizations.

Partitioning	FPGA	DSP	Estimated Power	costs
HW	XC4VFX40	-	957 mW	\$ 526
HWSW1	XC4VFX40	TMS320C5509	993.5 mW	\$ 541.08
	XC4VFX40	TMS320C6745	1278 mW	\$ 537.38
	XC4VFX40	TMS320C6747	1267 mW	\$ 539.03
HWSW2	XC4VFX40	TMS320C6747	1280 mW	\$ 539.03
	XC4VFX40	TMS320C6745	1291 mW	\$ 537.38
HWSW3	XC4VFX20	TMS320C6747	934 mW	\$ 342.19
	XC4VFX20	TMS320C6745	945 mW	\$ 340.54

B.5 Design Effort Estimation

In this Section further details regarding the design effort estimation presented in Sec. 5.5 are shown. The different Tables show the estimated and measured effective lines of code (eLOC) of different steps of the design space exploration. This information forms the basis for the design effort estimation shown comparing the Tripartite Design Approach (TDA) with a traditional approach. Tab. B.22 compares the eLOC of both approaches for the whole design space exploration.

Table B.10: Effective lines of code of different communication elements.

Library Element	eLOC	Other Elements	eLOC
Blocking Read	18	Local Link Interface	27
Nonblocking Read	19	Phase Diff	22
Blocking Write	19	Resampler FIFO	40
Nonblocking Write	25	Time Diff	28
Sum	96	Sum	117

Table B.11: Effective lines of code of the computation elements. Additionally, all lines related to a complex data structure (not Array or Const Array) are listed as CTL. Finally, the lines of code of each module without the CTL calls are listed.

Computation Element	eLOC	CTL Calls	eLOC without CTL
RTP Encode	101	9	92
G711 Encode	37	0	37
Jitter	29	0	29
Clock Recovery	44	0	44
G711 Decode	17	0	17
Resampler	71	13	58
Sum	299	22	277

Table B.12: Number of calls of each used member function of the Queue in the RTP Encode unit and eLOC of these functions are listed. "eLOC if inlined" denotes the data structure related eLOC if these functions are inlined.

Member Function	# of Calls	eLOC	eLOC if inlined
Instantiation	1	2	2
`clear()`	1	3	3
`size()`	1	1	1
`push()`	2	4	8
`pop()`	2	4	8
`front()`	2	4	8
Sum	9	18	30

Table B.13: Number of calls of each used member function of the Deque in the Resampler unit and eLOC of these functions are listed. "eLOC if inlined" denotes the data structure related eLOC if these functions are inlined.

Member Function	# of Calls	eLOC	eLOC if inlined
Instantiation	1	2	2
`clear()`	1	3	3
`resize()`	1	3	3
`push_front()`	1	6	6
`pop_back()`	1	5	5
`read()`	2	1	2
Sum	7	20	21

Table B.14: Design effort distribution among communication, computation and data structures. "All Lines of Code" denote the eLOC of all communication and computation elements and complex data structures. "Traditional" and "TDA" compare the eLOC a designer has to write in both approaches using a data structure and a communication library.

	All Lines of Code	TDA	Traditional
Communication	213	117	117
Complex Data Structures	51	22	22
Computation	253	277	253
Sum	517	416	392

Table B.15: To form synthesizable modules out of the HWSW_Modules, top level modules as described in Sec. 4.1.3.1 have to be built. eLOC of these modules is listed in this table for all computation elements.

	RTP Enc.	Resampler	G711 Dec.	G711 Enc.	Jitter	CLK Rec.	Sum
eLOC	60	42	26	26	24	38	216

Table B.16: Effective lines of code of different refinement steps towards a synthesizable hardware module using the traditional approach.

RTP Enc.	Resampler	G711 Dec.	G711 Enc.	Jitter	CLK Rec.	Sum
Data type refinement						
1	11	0	0	0	6	18
Remove complex data structures						
30	21	0	0	0	0	51
Change process type and add clock and reset						
6	6	6	6	6	6	36
Adding I/O protocol						
82	16	28	28	28	28	210
119	54	34	34	34	40	315

Table B.17: Communication elements, which are application specific and therefore not in a communication library have to be refined for hardware synthesis. These steps have to be performed anyway independent of the chosen design approach. The design effort for these elements is listedin eLOC

Local Link Interface	Resampler FIFO	Phase Diff	Time Diff	Sum
17	39	54	17	127

Table B.18: Design effort in eLOC of the different data structure to memory structure mappings using the traditional and the Tripartite Design Approach.

	RTP Enc.	Resampler	G711 Dec.	G711 Enc.	Jitter	CLK Rec.	Sum
Traditional	3	3	1	0	0	0	7
TDA	3	3	1	0	0	0	7

Table B.19: Design effort in eLOC of moving the Resampler to software using the traditional and the Tripartite Design Approach.

Traditional design approach				
Remove complex data structures	Refine data types	Remove SystemC constructs	Add synchronization facilities	Sum
21	8	8	3	40

Tripartite Design Approach	
Port Mapping	4

Design effort for both approaches			
SW Resampler FIFO	SW Phase Diff	Hardware EMIF adapter	Sum
24	24	55	103

Table B.20: Design effort in eLOC of moving the G711 Encoder and Decoder to software using the traditional and the Tripartite Design Approach.

Traditional design approach		
Remove SystemC constructs	Add synchronization	Sum
12	3	15

Tripartite Design Approach	
Port Mapping	6

Design effort for both approaches		
Hardware EMIF adapter	Software EMIF adapter	Sum
6	10	16

Table B.21: Design effort in eLOC of moving the RTP Encode unit to software using the traditional and the Tripartite Design Approach.

Traditional design approach				
Remove complex data structures	Refine data types	Remove SystemC constructs	Add synchronization facilities	Sum
30	0	18	6	54

Tripartite Design Approach	
Port Mapping	7

Design effort for both approaches		
Hardware EMIF adapter	Software EMIF adapter	Sum
59	23	82

Table B.22: The Table compares the design effort in terms of effective lines of code of the traditional and the Tripartite Design Approach listing all three major design steps of the design space exploration.

Design step	Design effort in eLOC	
	TDA	Traditional
High Level design	416	392
Refinement to hardware	343	442
Moving elements to software	225	317
Sum	984	1151

List of Figures

List of Tables

Literature

[AFMS07] N. Abdelli, A.-M. Fouilliart, N. Mien, and E. Senn. High-Level Power Estimation of FPGA. In *IEEE International Symposium on Industrial Electronics (ISIE 2007)*, 2007.

[AG04] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Addison-Wesley Professional, 2004.

[AGSS09] S. Asodi, S.V. Ganesh, E. Seshadri, and P.K. Singh. Evaluation of transport layer protocols for voice transmission in various network scenarios. In *Applications of Digital Information and Web Technologies, 2009. ICADIWT '09. Second International Conference on the*, pages 238–242, August 2009.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.* The Kluwer Academic Press, 1997.

[BD04] D.C. Black and J. Donovan. *SystemC From The Ground Up.* Springer, New York, NY, USA, May 2004.

[BHG10] P. Brunmayr, J. Haase, and C. Grimm. A Tripartite System Level Design Approach for Design Space Exploration. In *Proceedings of the 2010 Forum on specification & Design Languages (FDL 2010)*, pages 50–55, September 2010.

[BHG11] P. Brunmayr, J. Haase, and C. Grimm. A Hardware/Software Codesign Template Library for Design Space Exploration. In *Proceedings of the 2011 Electronic System Level Synthesis Conference (ESLSyn 2011)*, pages 5–10, June 2011.

[BHS09] P. Brunmayr, J. Haase, and F. Schupfer. Late Hardware/Software Partitioning by using SystemC Functional Models. In *Proceedings of the 3rd Asia International Conference on Modelling and Simulation (AMS 2009)*, pages 194–199, May 2009.

[BR01] M. Burns and G. W. Roberts. *An Introduction to Mixed-Signal IC Test and Measurement.* Oxford University Press, 2001.

[BWH09] P. Brunmayr, H.D. Wohlmuth, and J. Haase. An Efficient FPGA Implementation of an Arbitrary Sampling Rate Converter for VoIP. In *Austrochip 2009*, pages 33–38, October 2009.

[CGM+09] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. In *Formal Methods in Computer-Aided Design (FMCAD 2009)*, 2009.

[CM08] P. Coussy and A. Morawiec, editors. *High-Level Synthesis from Algorithm to Digital Circuit.* Springer, Laussanne, Suisse, 2008.

[CMGT09] P. Coussy, M. Meredith, D. D. Gajski, and A. Takach. An introduction to high-level synthesis. *IEEE Design and Test of Computers*, 26(4):8–17, August 2009.

[CR83] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing.* Prentice Hall, 1983.

[DGP$^+$08] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski. System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems*, 2008:13, 2008.

[EJL$^+$03] J. Eker, J.W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity-The Ptolemy Approach. *Proceedings of the IEEE*, 91:127 – 144, 2003.

[ETS90] ETSI. Global System for Mobile Communications (GSM). website, 1990. `http://www.etsi.org/WebSite/Technologies/gsm.aspx`.

[FCM$^+$07] L. Filion, M. A. Cantin, L. Moss, E. M. Aboulhamid, and G. Bois. Space codesign: A SystemC framework for fast exploration of hardware/software systems. In *Design & Verification Conference and Exhibition (DVCON'07)*, San Jose, CA, 2007.

[FCP09] H. Fathi, S. S. Chakraborty, and R. Prasad. Voice over Internet Protocol. In *Voice over IP in Wireless Heterogeneous Networks*, pages 37–48. Springer Netherlands, 2009.

[FHT06] J. Falk, C. Haubelt, and J. Teich. Efficient Representation and Simulation of Model-Based Designs in SystemC. In *Forum on Specification and Design Languages 2006 (FDL2006)*, pages 129–134, September 2006.

[Fin10] M. Fingeroff. *High-Level Synthesis Blue Book.* Xlibris Corporation, 2010.

[GAGS09] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification.* Springer, 2009.

[GB07] M. Grant and B. Bailey. *ESL Design and Verification: A Prescription for Electronic System Level Methodologies.* Morgan Kaufmann, 2007.

[GD01] Daniel Gajski and Rainer Dömer. SpecC^{TM}. Website, 2001. `http://www.cecs.uci.edu/~specc/`.

[GHP$^+$09] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. Stefanov, D. D. Gajski, and J. Teich. Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, October 2009.

[GLMS02] T Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[Gra05] R. M. Gray. The 1974 origins of VoIP. *Signal Processing Magazine, IEEE*, 22(4):87–90, July 2005.

[Gup93] Rajesh Kumar Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems.* PhD thesis, Stanford University, 1993.

[GVNG94] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[Ham09] A. Hamann. *Iterative Design Space Exploration and Robustness Optimization for Embedded Systems.* PhD thesis, Braunschweig University of Technology, 2009.

[HD07] S. Hauck and A. DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation.* Academic Press, 2007.

[HPSV03] F. Herrera, H. Posadas, P. Sánchez, and E. Villar. Systemic Embedded Software Generation from SystemC. In *Design, Automation and Test in Europe Conference and Exposition (DATE 2003)*, pages 10142–10149, 2003.

[HR94] M. R. Headington and D. D. Riley. *Data abstraction and structures using C++.* Jones and Bartlett Publishers, Inc, 1994.

[HSKM08] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith. SystemCoDesigner: Automatic Design Space Exploration and Rapid Prototyping from Behavioural Models. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 580–585, June 2008.

[HT04] B. Hounsell and R. Taylor. Co-processor synthesis: a new methodology for embedded software acceleration. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 682 – 683, feb. 2004.

[IG08] IEEE and The Open Group. The Open Group Technical Standard Base Specification, Issue 7, 2008.

[Int80] International Telecommunication Union (ITU). Integrated Services Digtial Network. website, 1980. `http://www.itu.int/rec/T-REC-I/e`.

[Int90] International Telecommunication Union (ITU). G.711: Puls code modulation (PCM) of voice frequencies. Website, 1990. `http://www.itu.int/rec/T-REC-G.711/e`.

[Int02] International Telecommunication Union (ITU). G.712: Transmission performance characteristic of pulse code modulation channels. Website, 2002. `http://www.itu.int/rec/T-REC-G.712-200111-I/en`.

[Int03] International Telecommunication Union (ITU). G.114: One-way transmission time. Website, May 2003. `http://www.cs.columbia.edu/~andreaf/new/documents/other/T-REC-G.114-200305.pdf`.

[Int09] International Telecommunication Union (ITU). H.323 : Packet-based multimedia communications systems. Website, 12 2009. `http://www.itu.int/rec/T-REC-H.323/en/`.

[ISO94] ISO/IEC. ISO/IEC standard 7498-1:1994 Open Systems Interconnection - Basic Reference Model. website, 1994. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip`.

[ISO03] ISO/IEC. ISO/IEC 14882:2003(E) Programming Languages - C++. Website, 2003. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=38110`.

[JPPG04] D. Jonathan, J. Peters, J. Peters, and B. Gracely. *Voice over IP fundamentals.* Cisco Press, 2004.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP*, 1974.

[KAL11] T. Kempf, G. Ascheid, and R. Leupers. *Mulitprocessor Systems on Chip: Design Space Exploration.* Springer, 2011.

[Keh05] N. Kehtarnavaz. *Real-Time Digitial Signal Processing: Based on the TMS320C6000.* Elsevier, Burlington, MA, USA, 2005.

[KL93] A. Kalavade and E. A. Lee. A Hardware-Software Codesign Methodology for DSP Applications. *IEEE Design and Test of Computers*, 10(3):16–28, September 1993.

[KMN$^+$00] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1523–1543, 2000.

[Koo96] Ph. J. Jr. Koopman. Embedded System Design Issues (the Rest of the Story). In *Proceedings of the International Conference on Computer Design (ICCD 96)*, 1996.

[KS80] J. F. Kaiser and R. W. Schafer. On the Use of the I_0-Sinh Window for Spectrum Analysis. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 28:105–107, 1980.

[LK81] R. Lagadec and H.O. Kunz. A universal, digital sampling frequency converter for digital audio. In *IEEE Int. Conference on Acoustics, Speech and Signal Processing*, volume 6, pages 595 – 598, April 1981.

[LM00] H. Liu and P. Mouchtaris. Voice over IP signaling: H.323 and beyond. *Communica-*

tions Magazine, IEEE, 38(10):142 –148, October 2000.

[LPW82] R. Lagadec, D. Pelloni, and D. Weiss. A 2-Channel, 16-bit digital sampling frequency converter for professional digital audio. In *IEEE Int. Conference on Acoustics, Speech and Signal Processing*, volume 7, pages 93 – 96, May 1982.

[LZ74] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, 1974.

[Mar10] P. Marwedel. *Embedded System Design.* Springer, Dordrecht, The Netherlands, 2010.

[Noe05] T. Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers.* Newnes, 2005.

[NSD08] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27:542–555, 2008.

[Nut81] A. H. Nuttall. Some windows with very good sidelobe behavior. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 29(1):84–91, February 1981.

[oEI80] Institute of Electrical and Electronics Engineers (IEEE). IEEE 802.3 Ethernet Working Group. Website, 1980. `http://www.ieee802.org/3/`.

[oEI03] Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 1532-2002 Standard for In-System Configuration of Programmable Devices, 2003.

[oEI05] Institute of Electrical and Electronics Engineers (IEEE). IEEE 1800-2005 Standard for System Verilog- Unified Hardware Design, Specification, and Verification Language. IEEE standard, 2005. `http://ieeexplore.ieee.org/servlet/opac?punumber=10437`.

[OSB99] A.V. Oppenheim, R.W. Schafer, and J.R. Buck. *Discrete-Time Signal Processing.* Prentice Hall, 1999.

[OSC] OSCI. SystemC^{TM}. Website. `http://www.systemc.org`.

[OSC09] OSCI. SystemC Synthesizable Subset 1.3 draft. website, 2009. `http://www.systemc.org/members/download_files/check_file?agreement=systemc_synth_subset_1-3`.

[PYC06a] S. Park, S. Yoon, and S. Chae. A Mixed-Level Virtual Prototyping Environment for Refinement-Based Design Environment. In *7th IEEE International Workshop on Rapid System Prototyping*, pages 63–68, 2006.

[PYC06b] S. Park, S. Yoon, and S. Chae. Reusable Component IP Design using Refinement-based Design Environment. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 588 – 593, 2006.

[Ram84] T.A. Ramstad. Digital Methods for Conversion Between Arbitrary Sampling Frequencies. In *IEEE Trans. on Acoustics, Speech and Signal Processing*, volume 32, pages 577 – 591, June 1984.

[RFC80] User Datagram Protocol. Request for Comments: 768, August 1980. `http://www.faqs.org/rfcs/rfc768.html`.

[RFC81a] Internet Protocol. Request for Comments: 791, 1981. `http://www.faqs.org/rfcs/rfc791.html`.

[RFC81b] Transmission Control Protocol. Request for Comments: 793, September 1981. `http://tools.ietf.org/html/rfc793`.

[RFC02] SIP: Session Initiation Protocol. Request for Comments: 3261, 2002. `http://www.ietf.org/rfc/rfc3261.txt`.

[RFC03a] Real Time Control Protocol (RTCP) attribute in Session Description Protocol (SDP). Request for Comments: 3650, 2003. `http://www.ietf.org/rfc/rfc3605.txt`.

[RFC03b] RTP: A Transport Protocol for Real-Time Applications. Request for Comments:

3550, 2003. `http://www.ietf.org/rfc/rfc3550.txt`.

[RFC03c] RTP Profile for Audio and Video Conferences with Minimal Control. Request for Comments: 3551, July 2003. `http://www.ietf.org/rfc/rfc3551.txt`.

[Rus11] A. Rushton. *VHDL for Logic Synthesis.* Wiley, June 2011.

[SA05] P. Saint-Andre. Streaming XML with Jabber/XMPP. *IEEE Internet Computing*, 9:82–89, 2005.

[SA07] P. Saint-Andre. Jingle: Jabber Does Multimedia. *Multimedia, IEEE*, 14(1):90 –94, January - March 2007.

[San09] B. Santo. 25 Microchips That Shook the World. *IEEE Spectrum*, 46:34–43, 2009.

[Sch98] H. Schildt. *C++: The Complete Reference.* Osborne McGraw-Hill, 1998.

[SG84] J. O. Smith and P. Gosset. A flexible sampling-rate conversion method. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 19.4.1 – 19.4.2, San Diego, March 1984.

[Sim10] P. Simpson. *FPGA Design: Best Practices for Team-based Design.* Springer, 2010.

[SK94] S. Sengupta and C. P. Korobkin. *C++ Object-Oriented Data Structures.* Springer, 1994.

[SP94] E. Stoy and Z. Peng. An Integrated Modelling Technique for Hardware/Software Systems. In *IEEE International Symposium on Circuits and Systems (ISCAS'94)*, London, UK, 1994.

[SR00] H. Schulzrinne and J. Rosenberg. The Session Initiation Protocol: Internet-centric signaling. *Communications Magazine, IEEE*, 38(10):134 –141, October 2000.

[SSDM01] L. Semeria, K. Sato, and G. De Micheli. Synthesis of Hardware Models in C with Pointers and Complex Data Structures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):743–756, December 2001.

[STG$^+$01] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, August 2001.

[uAS07] Zain ul Abdin and B. Svensson. A Study of Design Efficiency with a High-Level Language for FPGAs. In *Parallel and Distributed Processing Symposium (IPDPS 2007)*, 2007.

[Val06] J. W. Valvano. *Embedded Microcomputer Systems: Real Time Interfacing.* CL-Engineering, 2006.

[WCHL09] C. Wu, K. Chen, C. Huang, and C. Lei. An Empirical Evaluation of VoIP Playout Buffer Dimensioning in Skype, Google Talk, and MSN Messenger. In *NOSSDAV'09 Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 97–102, June 2009.

[Wen11] M. Wenzl. Design and Implementation of an VoIP Stack for High Packet Rates in an FPGA. Master's thesis, Institute of Computer Technology, Vienna University of Technology, April 2011.

[XDS$^+$08] J. Xu, Y. Dou, J. Song, Y. Zhang, and F. Xia. Design and Synthesis of a High-Speed Hardware Linked-List for Digital Image Processing. In *Congress on Image and Signal Processing, CISP*, 2008.

[ZOS00] W. Zhao, D. Olshefski, and H. Schulzrinne. Internet Quality of Service: an Overview. Technical report, Columbia University, February 2000.

Internet References

[1] Analog Devices Incorporated. http://www.analog.com. Accessed: 19/10/2011.

[2] Axilica FalconML. http://www.axilica.com/fm_rel_2_0.html. Accessed: 22/10/2011.

[3] Binachip ESLerate. http://www.binachip.com/eslerate. Accessed: 21/10/2011.

[4] Bluespec High Level Synthesis. http://www.bluespec.com/highlevelsynthesis/highlevelsynthesis.html. Accessed: 21/10/2011.

[5] Catapult C Synthesis Datasheet. www.calypto.com/datasheets/Calytpo_CatapultC_DS.pdf. Accessed: 02/11/2010.

[6] Codetronix Mobius. http://www.codetronix.com/Mobius/Mobius.html. Accessed: 21/10/2011.

[7] CoFluent Studio. http://www.cofluentdesign.com/index.php?page=home. Accessed: 22/01/2012.

[8] CriticalBlue Cascade. http://www.criticalblue.com/criticalblue_products/pdf/CascadeProductOverviewMay2007.pdf. Accessed: 22/10/2011.

[9] Datasheet: Crystal Clock Oscillators ACOL and ACHL. http://www.abracon.com/Oscillators/acol-achl.pdf. Accessed: 02/11/2010.

[10] Forte Design Systems. http://www.forteds.com. Accessed: 12/02/2010.

[11] Freescale Semiconductor Incorporated. http://www.freescale.com. Accessed: 19/10/2011.

[12] Frequentis AG. www.frequentis.com. Accessed: 03/02/2012.

[13] Impuls C CoDeveloper. http://www.impulseaccelerated.com/ImpulseFlyerV1.pdf. Accessed: 22/10/2011.

[14] Matlab. http://www.mathworks.com/products/matlab. Accessed: 12/02/2010.

[15] NEC Cyberworkbench. http://www.nec.com/global/prod/cwb/synthesis.html. Accessed: 22/10/2011.

[16] Platform Architect. http://www.synopsys.com/systems/architecturedesign/pages/platformarchitect.aspx. Accessed: 22/01/2012.

[17] Poseidon Desing Systems Triton Builder. http://www.poseidon-systems.com/builder.pdf. Accessed: 22/10/2011.

[18] SmartDSP Operating System. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=SmartDSP_OS. Accessed: 19/10/2011.

[19] SoC Designer. http://www.carbondesignsystems.com/. Accessed: 22/01/2012.

[20] Space Codesign. http://www.spacecodesign.com/. Accessed: 22/01/2012.

[21] Synopsys Synphony C Compiler. http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx. Accessed: 22/10/2011.

[22] Texas Instruments Incorporated. http://www.ti.com. Accessed: 30/09/2011.

[23] Xilinx AutoESL. http://www.xilinx.com/tools/autoesl.htm. Accessed: 22/10/2011.
[24] Xilinx Incorporated. http://www.xilinx.com. Accessed: 02/11/2010.
[25] Y Explorations eXCite. http://www.yxi.com/products.php. Accessed: 22/10/2011.
[26] PowerPC 405. https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores, 1998. Accessed: 24/08/2011.
[27] Jabber. www.jabber.org, 1999. Accessed: 17/08/2011.
[28] TLV320AIC23 Stereo Audio CODEC, Data Manual. http://focus.ti.com/lit/ds/symlink/tlv320aic23.pdf, May 2002. Accessed: 09/11/2010.
[29] Skype. www.skype.com, 2003. Accessed: 02/11/2010.
[30] LM4550 AC 97 Rev 2.1 Multi-Channel Audio Codec with Stereo Headphone Amplifier, Sample Rate Conversion and National 3D Sound. http://www.national.com/ds/LM/LM4550.pdf, May 2004. Accessed: 09/11/2010.
[31] ATS Ground Voice Network Implementation and Planning Guidelines. http://www.eurocontrol.int/eatm/gallery/content/public/library/ATS%20Ground%20Voice%20Network%20Implementation%20&%20Planning%20Guidelines%20Ed1-0.pdf, February 2005. Accessed: 10/08/2011.
[32] Voice Communication System Procurement Guidelines. http://www.eurocontrol.int/eatm/gallery/content/public/library/VCS%20Procurement%20Guidelines%20ed%202-0.pdf, February 2005. Accessed: 10/08/2011.
[33] Cadence C-To-Silicon Compiler Data Sheet. http://www.cadence.com/rl/resources/datasheets/c2silicon_ds.pdf, 2008. Accessed: 12/05/2011.
[34] ML405 Evaluation Platform User Guide. http://www.xilinx.com/support/documentation/boards_and_kits/ug210.pdf, March 2008. Accessed: 09/11/2010.
[35] 88E1111 Product Brief. http://www.marvell.com/products/transceivers/alaska_gigabit_ethernet_transceivers/Alaska_88E1111-002.pdf, March 2009. Accessed: 09/11/2010.
[36] TMS320C6455 Data Sheet. http://www.ti.com/lit/gpn/tms320c6455, April 2009. Accessed: 26/03/2010.
[37] Actel. Actel SmartFusion customizable System-on-Chip. http://www.actel.com/products/smartfusion/. Accessed: 12/11/2011.
[38] Altera. Logic Array Blocks and Adaptive Logic Modules in Stratix III Devices. http://www.altera.com/literature/hb/stx3/stx3_siii51002.pdf. Accessed: 20/10/2011.
[39] Altera. Nios II Processor. http://www.altera.com/devices/processor/nios2/ni2-index.html. Accessed: 12/11/2011.
[40] Altera. PowerPlay Power Analysis. http://www.altera.com/literature/hb/qts/qts_qii53013.pdf. Accessed: 22/10/2011.
[41] Altera. Design Debugging Using the SignalTab II Logic Analyzer. http://www.altera.com/literature/hb/qts/qts_qii53009.pdf?GSA_pos=3&WT.oss_r=1&WT.oss=signaltab, May 2011. Accessed: 29/10/2011.
[42] AutoESL. AutoPilot User Guide, May 2010.
[43] BCC Research. Embedded Systems: Technologies and Markets. http://www.bccresearch.com/report/embedded-systems-technologies-markets-ift016d.html, 2010. Accessed: 23/02/2012.
[44] Cadence. Cadence C-to-Silicon Compiler User Guide, June 2010.
[45] Community Research and Development Information Service (CORDIS). Embedded Systems: Facts and Figures. Website, July 2006. http://cordis.europa.eu/ist/embedded/facts_figures.htm.
[46] A. Devices. Estimating Power for ADSP-TS201S TigerSHARC® Processors. http://www.

analog.com/static/imported-files/application_notes/EE_170_ADSP_TS201S.pdf. Accessed: 31/10/2011.

[47] A. Devices. VisualDSP++ Development Software. http://www.analog.com/en/processors-dsp/blackfin/vdsp-bf-sh-ts/products/product.html. Accessed: 31/10/2011.

[48] Digi-Key. Product list of Digi-Key Corporation. http://search.digikey.com/scripts/DkSearch/dksus.dll, September 2011. Accessed: 30/09/2011.

[49] ForteDS. Cynthesizer Data Sheet. http://www.forteds.com/products/cynthesizer_datasheet.pdf. Accessed: 08/02/2010.

[50] ForteDS. Cynthesizer User's Guide, May 2011.

[51] T. Instruments. Code Composer Studio. http://www.ti.com/lsds/ti/dsp/support/dev_tool/ccs_overview.page. Accessed: 31/10/2011.

[52] A. Limited. Advanced RISC Machine (ARM). http://www.arm.com. Accessed: 14/01/2012.

[53] M Squared Technologies LLC. Source Code Size Metrics by the Resource Standards Metric Tool. http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics_narration.htm. Accessed: 12/02/2012.

[54] NXP Semiconductors. I2C-bus specification and user manual. http://www.nxp.com/documents/user_manual/UM10204.pdf. Accessed: 08/04/2012.

[55] O. M. G. (OMG). Unified Modeling Language. http://www.uml.org/, 2000. Accessed: 15/01/2012.

[56] Silicon Graphics International. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/. Accessed: 27/04/2011.

[57] J. Smith. Digital Audio Resampling Home Page. http://www-ccrma.stanford.edu/~jos/resample/, January 2002. Accessed: 12/02/2010.

[58] TI. DSP/BIOS Real-Time Operating System. http://www.ti.com/tool/dspbios. Accessed: 19/10/2011.

[59] TI. Power Spreadsheet for TMS320C6455. http://focus.ti.com.cn/cn/lit/an/spraae8b/spraae8b.pdf. Accessed: 29/04/2010.

[60] TI. TMS320C6000 DSP Multichannel Buffered Serial Port (McBSP). http://www.ti.com/lit/ug/spru580g/spru580g.pdf, December 2006. Accessed: 15/09/2011.

[61] TI. TMS320C6455 DSK Tech. Ref. http://c6000.spectrumdigital.com/dsk6455/v2/files/6455_dsk_techref.pdf, September 2006. Accessed: 29/04/2010.

[62] TI. TMS320C6000 DSP External Memory Interface (EMIF). http://focus.ti.com/lit/ug/spru266e/spru266e.pdf, April 2008. Accessed: 09/11/2010.

[63] TI. TMS320C6000 Optimizing Compiler v 7.0 User's Guide. http://www.ti.com/lit/ug/spru187q/spru187q.pdf, February 2010. Accessed: 13/10/2011.

[64] Xilinx. MicroBlaze Soft Processor Core. http://www.xilinx.com/tools/microblaze.htm. Accessed: 12/11/2011.

[65] Xilinx. Virtex-4 Family Overview. http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf. Accessed: 26/03/2010.

[66] Xilinx. Virtex-4 FPGA User Guide. http://www.xilinx.com/support/documentation/user_guides/ug070.pdf. Accessed: 20/10/2011.

[67] Xilinx. XPower Estimator User Guide. http://www.xilinx.com/support/documentation/user_guides/ug440.pdf. Accessed: 29/10/2011.

[68] Xilinx. Virtex-4 FPGA Embedded Tri-Mode Ethernet MAC User Guide. http://www.xilinx.com/support/documentation/user_guides/ug074.pdf, February 2010. Accessed: 09/11/2010.

[69] Xilinx. ChipScope Pro 13.1 Software and Cores. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/chipscope_pro_sw_cores_ug029.pdf`, March 2011. Accessed: 29/10/2011.

Curriculum Vitae

Name:	Peter Brunmayr
Address:	Universumstrasse 26/12
	1200 Vienna
	Austria

Education

since Nov. 2007	Vienna University of Technology, Austria
	Ph.D program at the Institute of Computer Technology
2005 - 2007	Vienna University of Technology, Austria
	Upgrade program for University of Applied Science graduates
2001 - 2005	University of Applied Science in Hagenberg, Austria
	Masters program: Hardware/Software Systems Engineering
1992 - 2000	Bundesrealgymnasium Steyr, Austria

Employment

since Jun. 2011	Frequentis AG, Austria
	R&D Hardware Engineer
2007 - 2011	Vienna University of Technology, Austria
	Assistant at the Institute of Computer Technology
Jul. - Sep. 2006	Carnegie Mellon University, Pittsburgh, USA
	Intern at the Department of Electrical & Computer Engineering
2005 - 2007	CD Laboratory for Design Methodology of Signal Processing
	FPGA Design Engineer: Development of a scaleable FFT IP-Core
2004 - 2005	CD Laboratory for Design Methodology of Signal Processing
	Diploma Student: Development of a digital predistortion testbed
Jul. - Sep. 2003	Philips Semiconductors Gratkorn GmbH
	Intern: Regression test development for smart cards

Interests

Football, Diving, Traveling, Climbing, Tennis, Snowboarding