Automated Rapid Prototyping Environment for Field Programmable Gate Arrays

von

Georg Brandmayr

DIPLOMARBEIT

eingereicht am
Fachhochschul-Diplomstudiengang

Hardware/Software Systems Engineering

in Wien

im Juli 2005

Betreut von:

Univ. Prof. Dr. Markus Rupp

Dipl. Ing. MSc. Gerhard Humer

Dipl. Ing. Dr. Thomas Müller-Wipperfürth

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Wien, am 30. Juni 2005

Georg Brandmayr

All our science, measured against reality,
is primitive and childlike — and yet it is
the most precious thing we have.

Albert Einstein (1879-1955)

Acknowledgements

It is a pleasure to thank the many people making this thesis possible.

First of all my gratitude goes to my supervisors Dr. Markus Rupp, Gerhard Humer and Dr. Thomas Müller. Without Markus Rupp this thesis wouldn't exist, since the prototyping system was his idea and he provided a seat in the universities MIMO (Multiple-Input Multiple-Output) system lab. Gerhard Humer also enabled this thesis by financing it, providing the prototyping hardware platform and ideas for enhancements. Certainly Thomas Müller gets not fewer thanks for his guidance in writing this thesis and teaching me computer architecture.

I am indebted to my many student colleagues for providing a fun environment in which to learn and grow. I wish to especially thank Andreas Schröck, David Hangweyrer, Paul Hager and Robert Mayr for the great time in Hagenberg. Further I am grateful to my colleagues in the MIMO lab: Peter Brunmayr for joining my coffee addiction and Christian Mehlführer for useful hints. My colleagues at ARCS, Florian Kaltenberger and Florian Schupfer, get my thanks for useful hints and testing the prototyping flow. Very special thanks to Roland Lieger for his great efforts in developing board software and carefully proofreading this thesis.

I owe a great deal to my friends Markus Seckellehner, Albin Ebetshuber, Patrick Fastmann, Daniela Aichinger and Robert Netsch for cheering me up and taking me back to reality, when I was totally clouded by my work.

Lastly, and most important, I wish to thank my family for the loving environment and especially my grandparents for always truly supporting me.

Foreword

In January 2005 I started working on my diploma thesis at the Vienna University of Technology, full of enthusiasm. Yet there was some concern whether I am able to develop the adequate concepts and to finish timely in June, since the subject is quite big. Building an automated rapid prototyping system requires thorough understanding of both hardware design and software design, because tools have to be created allowing for automated conversion of algorithms into hardware designs. Fortunately the education in Hardware/Software Systems Engineering provided me with most basics in order to tackle the enormous challenge and finish in time.

For students planning to write their own thesis I recommend the really excellent, general tips on thesis writing from Steve Easterbrook [7]. Readers having questions on the thesis' details can contact me by email: `georg.brandmayr@fh-hagenberg.at`.

Now I am really happy that all efforts led finally to a helpful tool for designers using Simulink — helpful in particular for those intending to target field programmable gate arrays for prototyping in digital signal processing without using hardware description languages.

This document reports in detail on the prototyping system I developed in the last months. Hence, readers intending to use the system will find it useful. Moreover, Chapter one, two and four might also be interesting for readers with a general concern in (automated) rapid prototyping or those intending to build such a tool as well.

Abstract

Verifying DSP (digital signal processing) systems is an error-prone and time-consuming process, since manual steps are involved in the creation of prototypes. Rapid prototyping is an emerging key design methodology, allowing DSP system designers to quickly verify their algorithms by co-simulation in order to identify implementation-connected real-time problems. The presented FPGA (field programmable gate array) rapid prototyping system fully automates the creation of hardware prototypes defined by just a single input source in Generic-C. Hence it replaces manual HDL (hardware description language) creation. Prototypes are generated for a Xilinx Virtex-II FPGA, part of the RTS-DSP (real time simulation digital signal processing) board from ARC Seibersdorf research. The MathWorks' Simulink serves as platform for simulation and FPGA prototype co-simulation. Thus it accesses this prototype via an also automatically generated hardware-in-the-loop block. An SRRC (square root raised cosine) transmit filter in co-simulation, taken from the wireless field, successfully demonstrates the potential of the rapid prototyping system. Compared to manually coded prototypes the time required for creation of a prototype is reduced by at least one order of magnitude. Although area and speed rates of generated prototypes are typically worse than for manually coded prototypes the presented tool allows for automated generation of FPGA prototypes for co-simulation without requiring HDL coding.

Kurzfassung

Die Verifikation von digitalen Signalverarbeitungssystemen ist fehleranfällig und zeitraubend, da manuelle Schritte zur Erzeugung von Prototypen erforderlich sind. Rapid Prototyping ist eine Schlüsseltechnologie, die Entwicklern dieses Verifizieren ihrer Algorithmen anhand von Co-Simulation in kürzester Zeit erlaubt. Dadurch können Probleme im Zusammenhang mit der Echtzeitumsetzung frühzeitig erkannt werden. Durch das vorgestellte Rapid Prototyping System wird die Erzeugung von FPGA (Field Programmable Gate Array) Prototypen vollständig automatisiert. Da nur ein Quellcode in Generic-C erforderlich ist, wird manuelles HDL (Hardware Description Language) Portieren unnötig. Ein Xilinx Virtex-II FPGA, integriert in der RTS-DSP (Real Time Simulation Digital Signal Processing) Baugruppe von ARC Seibersdorf research, dient zur Realisierung der FPGA Prototypen. Simulink von The MathWorks ist der Simulator für die Co-Simulation von Prototypen, wobei auf diese durch ebenfalls automatisch erzeugte „Hardware in the Loop“ Blöcke zugegriffen wird. Ein Beispiel aus dem Gebiet der digitalen Signalverarbeitung in Co-Simulation, ein SRRC (Square Root Raised Cosine) Sendefilter, demonstriert die Leistungsfähigkeit des Rapid Prototyping Systems. Verglichen mit einem nach herkömmlicher, manueller Weise erzeugten Prototyp wird die benötigte Zeit für dessen Herstellung um mindestens eine Größenordnung reduziert. Obwohl die Werte für Flächenverbrauch und Taktung, im Vergleich zu handoptimierten Prototypen, typischerweise schlechter sind, ermöglicht das vorgestellte Rapid Prototyping Werkzeug die automatisierte Erzeugung von FPGA Prototypen und deren Integration in Simulink Modelle auf Knopfdruck.

Contents

List of Abbreviations

3G · · · · · · ·	Third Generation
A|RT · · · · ·	Algorithm to Register Transfer
AMBA · · · ·	Advanced Microprocessor Bus Architecture
API · · · · · ·	Application Programming Interface
ANSI · · · · ·	American National Standards Institute
ARC · · · · · ·	Austrian Research Centers
ARCS · · · · ·	Austrian Research Centers Seibersdorf
ASCII · · · · ·	American Standard Code for Information Interchange
ASIC · · · · ·	Application Specific Integrated Curcuit
BLAST · · · ·	Bell LAbs layered Space-Time
BRAM · · · ·	Block Random Access Memory
CD-ROM · · ·	Compact Disc Read Only Memory
CDFG · · · · ·	Control/Data Flow Graph
CLB · · · · · ·	Configurable Logic Block
COSSAP · · ·	COmmunications System Simulation Analysis Package
CPU · · · · · ·	Central Processing Unit
CRLF · · · · ·	Carriage Return Line Feed
DCM · · · · ·	Digital Clock Mangager
DLL · · · · · ·	Dynamic Link Library
DSP · · · · · ·	Digital Signal Processing or Digital Signal Processor
DVD · · · · ·	Digital Versatile Disc
EBNF · · · · ·	Enhanced Backus Naur Form
EDA · · · · · ·	Electronic Design Automation
EMIF · · · · ·	External Memory InterFace
FEC · · · · · ·	Fast Ethernet Controller
FFT · · · · · ·	Fast Fourier Transform

FIFO · · · · ·	First-In First-Out
FIR · · · · · ·	Finite Impulse Response
FPGA · · · · ·	Field Programmable Gate Array
FSM · · · · · ·	Finite State Machine
GCC · · · · ·	Gnu C Compiler
GSM · · · · ·	Global System for Mobile Communications
HDL · · · · · ·	Hardware Description Language
HIL · · · · · ·	Hardware In the Loop
HW · · · · · ·	Hardware
I/O · · · · · ·	Input/Output
ID · · · · · · ·	IDentity
IOB · · · · · ·	Input Output Buffer
IC · · · · · · ·	Integrated Circuit
IP · · · · · · ·	Intellectual Property
IR · · · · · · ·	InterRupt
ISE · · · · · ·	Integrated Software Environment
ISO · · · · · ·	International Standards Organization
JTAG · · · · ·	Joint Test Action Group
MAC · · · · ·	Multiply Accumulate
MEX · · · · ·	MATLAB EXecutable
MUX · · · · ·	MUltipleXer
MIMO · · · ·	Multiple-Input Multiple-Output
LMS · · · · · ·	Least Mean Square
LTI · · · · · ·	Linear Time Invariant
LVDS · · · · ·	Low Voltage Differential Signaling
OS · · · · · · ·	Operating System
OSI · · · · · ·	Open System Interconnection
PC · · · · · · ·	Personal Computer
PDF · · · · · ·	Portable Document Format
PLL · · · · · ·	Phase Locked Loop
PNG · · · · ·	Portable Network Graphics
RAM · · · · ·	Random Access Memory
RF · · · · · · ·	Radio Frequency
ROM · · · · ·	Read Only Memory
RPT · · · · · ·	Rapid ProtoTyper
RS-232 · · · ·	Recommended Standard-232
RTL · · · · · ·	Register Transfer Level

RTS-DSP · · ·	Real Time Simulation Digital Signal Processing
SDRAM · · ·	Synchronous Dynamic Random Access Memory
SFTP · · · · ·	Secure File Transfer Protocol
SMS · · · · · ·	Short Message Service
SOC · · · · · ·	System On a Chip
SODIMM · · ·	Small Outline Dual In-line Memory Module
SRAM · · · ·	Static Random Access Memory
SRRC · · · · ·	Square Root Raised Cosine
SSH · · · · · ·	Secure SHell
STL · · · · · ·	Standard Template Library
SW · · · · · ·	Software
Tcl/Tk · · · ·	Tool Command Language / Tool Kit
TCP · · · · · ·	Transmission Control Protocol
TI · · · · · · ·	Texas Instruments
UDP · · · · ·	User Datagram Protocol
VHDL · · · · ·	Very high speed integrated curcuits (VHSIC) Hardware Description Language
VC++ · · · ·	Visual C++
VS · · · · · · ·	Visual Studio
WD · · · · · ·	Working Directory
XST · · · · · ·	Xilinx Synthesis Tool
μC · · · · · · ·	*Micro* Controller

Chapter 1

Introduction

1.1 Motivation

Trends in the electronic industry show that design complexity is increasing faster than ever before, although the time for development of a product is getting even smaller. Consider, for example, mobile phones in the wireless communications market. In the early 90's, when the Global System for Mobile Communications (GSM) was launched, the architecture of a mobile was straightforward [9]. A typical product consisted of numerous small integrated circuits, with roughly equal partition between radio frequency (RF) and digital frontend. Besides phone calls, and soon thereafter, short message service (SMS), no substantial features were provided. However, today's architectures of third generation (3G) devices show an above average complexity increase in the digital part, because the market demands as much functionality as possible in a single device. System-on-a-chip (SOC) helps reduce the number of integrated circuits (ICs) and offers unprecedented complexity — complexity which designers have to deal with.

Prototypes can help unveil design bugs, which might be hidden in the simulation stage, since they allow exploring the behavior of a "real" product. In particular FPGA prototypes are well suited to explore the behavior of hardware (HW) components, since FPGAs have already moved beyond their traditional applications into new domains such as digital signal processing. The emergence of multimillion-gate, "application specific integrated circuit (ASIC)-like" devices incorporating innovative memory architectures and embedded processors enables even the implementation of complex algorithms [19]. FPGA prototypes allow designers to estimate parameters, which are typical for design portions to be implemented in an ASIC or FPGA, like real time algorithms with extensive, repetitive multiplications at speeds of some 100 MHz. One such parameter is the area consumption of the component, which is strongly influenced, besides design complexity, by the utilization of FPGA-specific hard macros, such as multipliers, block random access memories (BRAMs) or DSP-slices. Even timing issues, such as long critical paths, can be analyzed, because they will not be much different in the final product. Thus, it is possible to identify bugs in an early project stage for typically much less cost than in a product's final stage. Another reason for building a prototype is to convince potential customers of the capabilities of the product, which might be far from completion. From these points of view, prototypes seem to be the perfect solution for all problems. Nonetheless, a drawback of prototypes is that their implementation is costly and time consuming for today's high complexity systems. There is no use for a prototype if the labor to build it equals building the product itself or when it becomes available after the product release. Additionally, decreasing time-to-market forces designers to skip work not of utmost necessity for product completion, which includes prototypes. Traditionally, FPGA prototypes are implemented manually, which means that a hardware designer obtains a behavioral description of the prototype, for instance a MATLAB script, and tries to convert it into an HDL description. This is the same procedure as for the product itself, which results in the aforementioned handicaps of prototypes. Nevertheless, designers can greatly benefit from prototypes if their creation is easier to deal with.

Automated rapid prototyping, on the other hand, provides all advantages of prototyping, but avoids its disadvantages. The error prone, manual creation of prototypes is replaced by an automated design flow, which uses an algorithm to generate the hardware description from the behavioral description, referred to as behavioral synthesis. This automated approach prohibits human errors from VHDL[1] porting and speeds up the prototype creation time by powers of ten, due to automation. Thus, rapid prototyping does not constrict designers in keeping tight time-to-market restrictions. In addition, the substantial savings in time even allow for iterations in the prototyping flow, which are useful in gradually improving the design. Some of the aspects presented so far are not new. The five ones approach, introduced by Rupp in [27], recommends using one "golden" code as well as heavy tool automation. In [30] a rapid prototyping environment, very similar to mine, is depicted except that the one source paradigm is not used, since VHDL code is created manually.

As one may have figured out by now, the presented approach provides another advantage: designers do not have to (immediately) deal with HDL any longer. However, this is only true as long as there are no bugs in the generated hardware. In such cases the HDL must be debugged, which still has to be done by hardware experts. Yet, compared to the situation some years ago, today's designers have valuable support in simulating hardware (more precisely HDL) by high level tools. For instance, Simulink has been

[1]very high speed integrated circuits (VHSIC) hardware description language.

recently extended with the award winning[2] capability to simulate HDL code directly in a simulation model. Thus, it is now possible to use all high level features of Simulink, like data representation on scopes, for handling and visualizing HDL simulation data. For example, the input for a VHDL filter could be generated by a Simulink sweep generator, applied to the VHDL entity and the outputs can be viewed on a time scope or fast Fourier transform (FFT) scope — features not provided by common HDL simulators. Again it is not necessary to deal directly with the HDL to perform co-simulation. Nevertheless, one should not get the impression that such an automated rapid prototyping flow renders the knowledge of hardware (description languages) unnecessary. The automated flow still requires these competencies, even when their application is not required in the same extent as for manual implementation.

1.2 Ambition

Due to all the aforementioned benefits of rapid prototyping this thesis is aimed at developing an FPGA rapid prototyping environment for the application area of DSP. This thesis' work has been previously published in [4], which provides an overview, omitting details addressed here.

Typically a DSP system's architecture is built by using block oriented system level simulation tools, such as Synopsys' CoCentric System Studio or The MathWorks' Simulink. The prototyping environment must create an FPGA prototype and provide a seamless integration into the block based simulator in form of a ready-to-use "hardware in the loop" (HIL) block. This block, accessing the FPGA prototype, is then used for co-simulation in existing simulation models. The prototype's desired behavior is specified in a high-level level, C like, programming language. Highest priority is given to full automation of the prototyping flow, in order to minimize prototyping time and to keep a *single* source, as suggested in [27]. This means that even the low level HW descriptions have to be created automatically, demanding for behavioral synthesis of the single source.

Designers, familiar with HW design, know that creating an FPGA design requires time — computing time alone may take hours. In case of an error redesign will consume nearly the same amount of time. The better part of this time can be saved by first creating behavioral prototypes — an additional task for the automated rapid prototyping system.

Eventually fixed-point numerics must be supported, since most embedded DSP systems do not allow for floating-point numerics due to HW area and power consumption restrictions.

[2]The Mathworks and Mentor Graphics won the EDN Innovation of the Year Award for their co-simulation environment "Link for ModelSim." This award honors outstanding electronic products [8].

1.3 Outline

The following chapters introduce the automated FPGA rapid prototyping concept and the upcoming tool: Rapid ProtoTyper (RPT). I recommend reading all chapters in the given sequence, since most of them are constitutive on each other and contain references on previous chapters. This applies in particular to Chapter 5, which sets up directly on Chapter 4.

Chapter 2 provides an overview of co-simulation, common implemented and imaginable FPGA rapid prototyping approaches and C to register transfer level (RTL) conversion concepts.

The platform comprising the prototype carrier FPGA, that is the ARCS RTS-DSP board,[3] the block based simulator and major, already initially appointed design tools are presented in Chapter 3.

Chapter 4 presents key concepts and fundamentals developed in order to implement the rapid prototyping tool. The simulator's role will be explained in detail, followed by methods used to integrate prototypes in co-simulation. These methods include analysis of the single input source, leading to creation of the FPGA prototype and required simulation software.

Chapter 5 identifies detailed, concrete prototyping tasks and presents their implementation in the realized design flow, based upon the ideas introduced in Chapter 4. Design flow elements, such as programmed tools and libraries, are described in detail. Finally the graphical user interface (GUI), integrating all tools in an application, is introduced.

Chapter 6 demonstrates the prototyping environment's application on a practical example. A typical wireless field simulation model is used for co-simulation with an automatically generated behavioral prototype and an HIL block, generated by application of the prototyping flow on the model's transmit filter.

Finally Chapter 7 gives a summary on the work, showing its limitations and capabilities. In addition future optimizations, such as integration of better tools, are suggested.

Tool installation instructions and notes are listed in Appendix A. The tool structure of RPT is described in detail in Appendix B. Appendix C lists the contents of the CD-ROM, comprising the diploma thesis PDFs, bibliography items and the example presented in Chapter 6.

[3] Austrian Research Centers Seibersdorf Real Time Simulation Digital Signal Processing board.

Chapter 2

FPGA Rapid Prototyping

This chapter provides an overview of common implemented and imaginable FPGA rapid prototyping approaches, C to RTL conversion concepts and co-simulation.

2.1 Block-Based Co-Simulation

Co-simulation is typical for verifying prototypes and model based simulation is typical for architectural design. This section states the motivation for block based co-simulation.

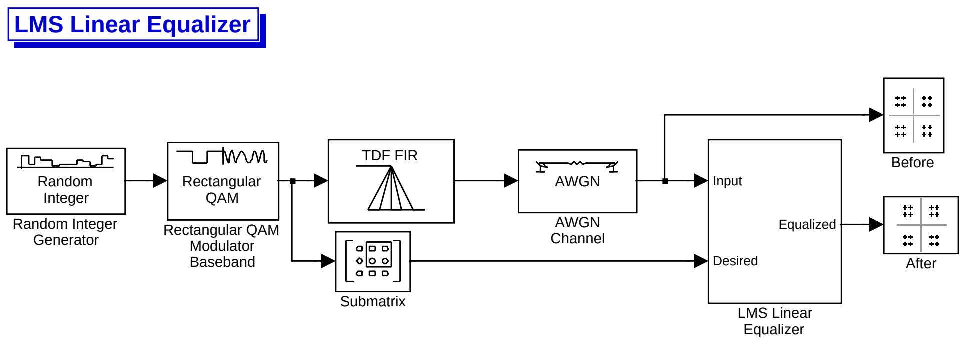


Figure 2.1: Example of a block based Simulink model.

Imagine a rather complex DSP simulation model in a graphical, block based simulation environment like the widely known Simulink, e.g., the MIMO BLAST (Bell Labs layered space-time) system implemented in [3] or the one depicted in Figure 2.1. At an early project stage, the entire simulation will execute solely on the personal computer (PC), since it is based on algorithmic library blocks. Once this algorithmic simulation succeeds one can move closer towards product implementation in HW. In order to verify the HW feasibility (for more reasons refer to Chapter 1), designers might

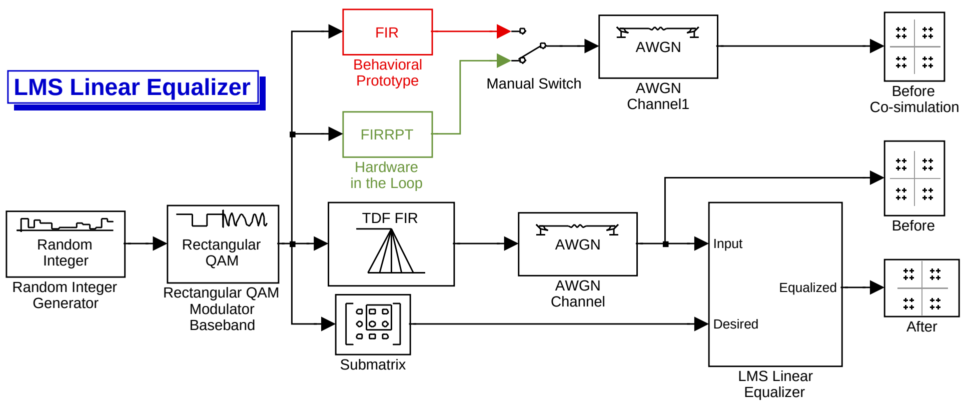


Figure 2.2: Example for co-simulation of prototyping blocks.

want to migrate portions of their design into FPGAs, which constitutes a characteristic case of operation for the rapid prototyping system. Hence, it is intended to replace a specific Simulink block, e.g., a subsystem of library blocks, by a block accessing an FPGA prototype executing the same function. However, first the prototype's correct function has to be approved by co-simulation, i.e., the prototype is simulated *along with* its "behavioral" counterpart — the original block. Block behavior for various scenarios can be verified by comparing both outputs.

It must be considered that creation of FPGA prototypes is time consuming, in particular for performing multiple runs when prototypes must be rebuilt. Solely the computational time required for creating a bitstream from HDL code is at least half an hour for complex designs. Hence it is useful to perform co-simulation with a behavioral prototype prior to creation of the FPGA bitstream. Behavioral prototype means a purely software (SW) based (PC-executed) model of the HW prototype, e.g., a VHDL co-simulation. The model in Figure 2.1 depicts an least mean square (LMS) linear receiver, taken from the wireless field, comprising a random source, modulator, transmit filter, channel and the said receiver. Assuming the designer co-simulates the transmit filter, an expanded model is shown in Figure 2.2. First the red behavioral prototype, thus the prototype's SW model, is added to the simulation. The constellation diagrams before the equalizer are used to verify the prototype. It is sufficient to compare results before the equalizer, since only the prototype's output is of interest, but not the equalizers behavior (in this co-simulation). Once co-simulation of the behavioral prototype leads satisfying results, the green hardware prototype can be generated and eventually co-simulated too, referred to as "hardware in the loop" (HIL) simulation. Verification is performed equally to the previous co-simulation of the behavioral prototype. A verification approach allowing for automation is to build the difference of the original filter output and its respective prototyping block output by means of an addition block, followed by checking if any sample is non-zero.

HW Prototypes are created by mapping a blocks functionality into an FPGA. Co-simulation is used to verify behavioral prototypes and HIL prototypes.

2.2 Prototyping Methodologies

Existing and imaginable FPGA prototyping methodologies for HIL simulations are discussed by analyzing their pros and cons. Unlike the previous section this section describes *how* FPGA prototypes are created.

In a top level view the purpose of the prototyping system, as demonstrated in the previous section, is to build a prototype represented in the simulation model by a single HIL block. This block accesses, via yet undefined interfaces (IFs), the prototype on the FPGA. The question arises, how this prototype is created. Several approaches exist, differing mainly in the front end, i.e., the input source. The back end is the same: HDL descriptions are synthesized, mapped, placed and routed, eventually resulting in a bitstream for FPGA configuration. Such FPGA implementation tools as Xilinx' Integrated SW Environment (ISE), introduced in Section 3.3.1, and Quicklogic's QuickWorks were originally developed to help designers effectively develop logic circuits, since that *was* the best fit for FPGAs.

Synthesis: HW synthesis means that RTL HDL descriptions are analyzed and converted into a netlist of logic elements, allowing for the generation of an ASIC or FPGA implementation. VHDL, suitable for synthesis, must contain specific patterns to allow *hardware inference*, i.e., the synthesis tool allocates hardware for these patterns. In general the hardware used for inference is all-purpose hardware, applicable for different tasks, such as look-up tables or flip-flops. Additionally most FPGAs provide optimized hardware blocks which are dedicated to particular tasks only, like block memories. Commonly these resources cannot be inferred from VHDL code but must allocated by *hardware instantiation*, which makes code technology dependent. If VHDL is intended to be generated for different platforms, this must be regarded, because each one will require its own instantiations while others will not offer some resources at all.

The following three sections describe how HDL code for FPGA implementation tools is created according to the different prototyping approaches.

2.2.1 Structural Synthesis

This prototyping approach means utilizing a model's structure by converting it into an associated HDL description suitable for synthesis. For each block in the simulation model exists an associated, generic HDL module, which is adapted to block parameters. Hence the simulation model itself is the input source to the rapid prototyping system. The models structural information is used to interconnect these HDL modules. This approach even allows for the use of extremely optimized or pre-synthesized cores, since only the interconnections for the modules are created and their generic's actual values are set. Xilinx' System Generator for DSP [32] was the first tool implementing this concept and bridged the gap between the world of the system engineer and the hardware designer.[1] By seamlessly integrating with Simulink, System Generator leveraged a powerful visual data flow environment suited for modeling digital algorithm signal flow graphs, and allowed the designer to generate bit- and cycle-accurate hardware implementation from the system model automatically [16].

Though being very promising this approach suffers from missing means of control, since block based models only allow to access interconnections as opposed to block internals. Building sophisticated control logic with the provided library blocks is very labor intensive and will, due to required workarounds, often result in suboptimal results. Nevertheless this is a comfortable solution suitable for algorithm designers requiring only library components.

2.2.2 Behavioral Synthesis

Another approach, providing more *flexibility* and better access to block *internals*, is to specify the input to the prototyping system, i.e., the block description, in a high level (programming) language, which is commonly referred to as behavioral synthesis. It interprets an algorithmic description of a desired behavior and creates hardware that implements that behavior. The description language must be widely accepted to avoid the tedious acquisition phase of a new language, suggesting to use C/C++. For rapid prototyping a prototype, replacing a simulation block, must be created. This implies that all block information, in addition to the description relevant to the simulator engine, must be contained in this source. Common programming languages, however, do not provide facilities for specifying Simulink items. Of course the language can be extended with proprietary constructs serving one's needs, but this prohibits compatibility to International Standards Organization (ISO) compliant compilers. It is very beneficial that behavioral prototype blocks, as mentioned in Section 2.1, can be created by compiling the description for using it as block the simulation model.

In behavioral synthesis several optimizations[2] are performed to reduce the algorithm's complexity and then the description is analyzed to deter-

[1]Comparable Simulink to FPGA tools are available by Synplicity, Lyrtech and Sundance.

[2]E.g., common subexpression elimination.

mine the essential operations and the dataflow dependencies between them. Allocation and binding algorithms assign high level operations to specific functional units such as adders, multipliers, comparators, etc. Finally, a state machine is generated that will control the resulting datapath's implementation of the desired functionality. The datapath state machine outputs are in RTL HDL, optimized for use with the FPGA implementation tools mentioned before. More details on behavioral synthesis are examined in Section 2.3.

Although tools exist behavioral synthesis is currently a research field on its own. Since C to RTL conversion is the method of choice for the rapid prototyping system it is discussed in Section 2.3. The biggest challenge still lies in the semantics of C/C++ to RTL conversion, i.e., how C/C++ constructs are mapped to HW elements.

2.2.3 Manual HDL Creation

The last, apparent approach is creating HDL code for the prototype manually, as used in [30].

Although manual HDL creation cannot be compared directly to the two other approaches, since it does not allow for a *single* source and in turn *automatic* prototyping, at least the remaining tasks for HIL simulations can be kept automated. However, VHDL designs can be immediately co-simulated as behavioral prototype by connecting the simulation platform to a HDL simulator, as performed in [30] for ModelSim and Simulink. Manual code creation enables full exploration of the FPGA architecture by using hard macros like BRAMs, which are not accessible by behavioral synthesis. Unfortunately the time required for prototype creation is significantly higher than for the two approaches presented above, in particular for complex designs since the HDL code gets complex.

Concluding, manual HDL creation is no good choice for automated rapid prototyping, because time consuming programming on RTL level is required.

2.3 C To RTL

Since a C-dialect is used as input to the prototyping system, C to RTL HDL conversion is required and analyzed in detail.

Digital HW design traditionally had the disrepute of requiring heavy efforts for minimal results. This was caused, to the bigger part, by outdated design methodology like full manual wire routing. Today's modern synchronous digital designs are synthesized from HDL code, minimizing design efforts compared to manual techniques. However, even this methodology reached its limits due to the ever increasing, enormous design sizes of 10MGates+. Designers struggle with complex HDL sources, demanding for new, system level design methodology. The electronic design automation

(EDA) industry, on the other hand, desperately tries to get C to RTL conversion tools ready for practical use, while still not adopted by most HW designers. In the following C to HDL conversion strategies are presented.

2.3.1 Strategies

A widely accepted hardware semantics of American National Standards Institute (ANSI) C/C++ (*not* SystemC) is still not defined.

Parallelism

Writing a program in C is a straight forward task, but describing a piece of hardware is much more difficult because a program is a *sequence* of operations, executed one after the other, while in hardware as in ASICs or FPGAs operations are also executed in *parallel*. Unlike VHDL, C does not provide any language mechanism to express parallelism. In order to take care of parallelism a new language construct can be introduced. E.g., Celoxica's Handel-C [6] uses `par { }` to tell the compiler about parallelism. Although this approach is successful[3] it suffers from incompatibility to the language standard due to the mentioned proprietary extensions. To avoid this problem either parallelism could be left completely to the C to RTL translation tool or it is denoted implicitly in the code by language compliant constructs.

Timed C++

In behavioral synthesis *timed* means that source code must contain a notion of time. Designers have to take care for scheduling their operations over cycles, sharing resources and inserting pipeline stages.

This notion of time is a syntactical construct, e.g., A|RT Builder [1] uses function calls to schedule operations in one cycle. All code in a single function call is mapped into HW operating in a single clock cycle. Registers are inserted by using special variables, e.g., those with a `static` storage class specifier. Thus long combinatorial logic can be pipelined by register insertion. To perform this functionality the tool can use a one to one mapping from C++ constructs into corresponding HDL output — every C++ code piece gets its HDL counterpart, as applied by A|RT Builder. A sequential process creates all registers, the C++ code sequence is translated into a VHDL code sequence and mapped into a combinatorial process. One might think now, that this methodology does not provide an abstraction gain, which is indeed true. However, the important fact is, that the source language is C++ and the source code size is smaller than the output HDL.

Concluded, the synthesis of timed C++ is not beneficial in terms of abstraction, since operation-scheduling etc. must be hard-coded in the source

[3]Handel-C tools are already over 10 years on the market.

with specific constructs, which is basically RTL modeling in C++.

Untimed C++

Synthesizing *untimed* C++ eliminates shortcomings from timed synthesis. The source code is no longer cycle true, i.e., it comprises only the algorithm rather than architectural information. More precisely, the source code contains only the information *what* operations have to be performed as opposed to the RTL-like style required in timed descriptions, describing *what and how* operations have to be performed.

First the source is analyzed syntactically and transformed into an internal data representation. Then the algorithm is optimized by constant propagation, eliminating common subexpressions, etc. Inputs, outputs, and operations of the algorithm are identified, and data dependencies between them are determined, resulting in a control/dataflow graph (CDFG). This determines which values are needed prior to computation of other values. Note, that still no concept of time exists. Resource allocation establishes a set of functional units, taken from the technology library, that will be suitable to implement the design. Eventually the concept of time is introduced by transforming the CDFG into a finite state machine (FSM) representation. Using the data dependencies of the algorithm and the latencies of the functional units in the library, the operations of the algorithm are assigned to specific clock cycles. Design constraints will determine the result with respect to latency, pipelining, and resource utilization. Binding assigns the operations of the algorithm to specific instances of functional units from the library. Values that are written in another cycle than they are read, are stored in registers, allocated by register binding. The datapath and FSM resulting from all of the previous steps are output as RTL source code in the target language [23].

Hence, synthesis from untimed code requires the tool to design an architecture for the source description. Since a large range of possible architectures exists, designers control the tool by applying constraints, such as maximum latency, the number of resources etc., to select their appropriate architecture.

2.3.2 Tool Comparison

Various tools are available for C/C++ to RTL conversion. I arbitrarily selected four to discuss them in the following.

DK Design Suite

DK Design Suite from Celoxica is one of the most successful[4] C based design flow tools on the market.

Celoxica developed its own C dialect Handel-C [6]. Handel-C is typical of the home-grown C-based simulation and synthesis languages developed by universities and EDA companies. It preserves traditional C syntax and control structures, making it easy for C programmers and hardware designers to understand. In addition to hardware-centric datatypes, Handel-C also includes proprietary extensions that facilitate dataflow representations and support parallel programming. This flow involves manually translating the untimed algorithm into Handel-C. Following verification via simulation (which requires Celoxica's compiler), the Handel-C representation is directly synthesized into a gate-level netlist.

Using a proprietary language means users cannot use alternative simulation or synthesis tools. As a result, many engineers prefer standards-based alternatives. Theoretically, manual translation of MATLAB to Handel-C should be relatively painless because Handel-C is close to pure C. In practice coercing Handel-C to adequately capture the design in a form suitable for the synthesis engine requires intensive work by an expert user.

All of the implementation "intelligence" associated with the design has to be hard-coded into the Handel-C, which therefore becomes implementation-specific and timed, as explained above. Furthermore, users have minimal control over the Handel-C synthesis engine, which is something of a "black box" to work with and which doesn't take advantage of the target technology. E.g., no account is taken of FPGA elements like multipliers and BRAMs. This implies some nonintuitive manipulation of the C code to achieve speed and size requirements. In short, design teams may end up taking as much time creating adequate Handel-C as they would hand-creating the RTL, thereby nullifying the advantage of C-based design flows [12].

Catapult-C

Catapult-C [21], one of the most recently released C-based design tools, creates "*optimized* ASIC/FPGA hardware from *untimed* C++," according to its developer Mentor Graphics. The two keywords here that suggest significant progress are optimized and untimed. The following words from [24] provide the idea of Mentor's approach:

> Most of the other C/C++ hardware generation tools have relied on pseudo-timed input with specialized libraries adapting C and C++ to hardware design by adding scheduling constraints and other hardware-specific information into the source description. Mentor's approach, by working from completely untimed

[4]Available for over 10 years and most used in the world, according to Celoxica.

algorithmic descriptions, gives the compiler the maximum flexibility in creating a hardware architecture that is optimized for the design goals of the project. It also means that C or C++ targeted at hardware is more like the generic code that a software developer would normally write.

Since the untimed source does not contain architectural information, a multitude of different architectures can be implemented, differing in area, speed, latency and others. Catapult-C offers the exploration of these architectures by plotting performance charts (micro-architecture what if analysis [19] in Mentor Graphic's terminology). Then designers select the alternative, that fits their constraints best.

A major problem in behavioral synthesis is the connection of algorithms to interfaces, e.g., an Advanced Microprocessor Bus Architecture (AMBA) bus. These interfaces impose constraints on the algorithms architecture and therefore must be considered in synthesis. However, most behavioral synthesis tools available do not. Catapult-C uses a patent-pending interface synthesis technology [19], that creates a wrapper around the algorithmic design and connects it to the external HW. Interface synthesis allows to switch between various external interfaces and explore results without modifying the source.

Catapult-C uses pure, untimed C++ and integrates interface specification. It clearly is the most advanced among today's behavioral synthesis tools.

CoDeveloper

CoDeveloper from Impulse Accelerated Technologies supports the rapid creation of hardware-accelerated systems using FPGAs and the C programming language.

CoDeveloper is a C language development system for coarse grained programmable hardware targets including mixed processor and FPGA platforms. CoDeveloper's core technology is the Impulse-C library and related tools that allow standard ANSI C to be used for the expression of highly parallel applications and algorithms targeting mixed hardware/software targets. This set of C libraries and tools (which include the CoBuilder RTL generator, CoMonitor Application Monitor and the CoDeveloper Application Manager) can be used in conjunction with standard programming environments including Microsoft's Visual Studio (VS), Gnu C Compiler (GCC) and other standard tools for the development, debugging and implementation of highly parallel applications directly onto programmable hardware platforms [11]. CoDeveloper performs loop unrolling, common subexpression elimination and automatic parallelizing when generating HW. Contrary to Handel-C no CoDeveloper uses no proprietary language extensions, except compiler pragmas for manual architecture manipulation.

I evaluated CoDeveloper for use in the rapid prototyping flow. The most remarkable thing is that designers must make heavy use of the mentioned Impulse-C library, if they want to migrate an algorithm to HW. This library provides interface (IF) elements such as “streams,” which must be used in algorithms targeted at FPGAs. More precisely, a C function targeted at FPGAs must only use Impulse-C library interfaces, which are mapped into corresponding HDL equivalents, e.g., a first-in first-out (FIFO) memory for the mentioned stream. Obviously the C code is strongly Impulse-C library dependant, since interfaces are included in the code, as opposed to Catapult-C. It is not possible to create HDL without these interfaces, making CoDeveloper only a second choice. In general the product seems to be targeted at C programmers with very little or no HW knowledge, who want to accelerate their software with ready-to-use FPGA power.

A|RT Builder

A|RT Builder[5] from Adelante Technologies is a design tool that translates a C++ based functional specification of an algorithm into an RTL HDL description. A|RT stands for Algorithm to Register Transfer. The input to A|RT Builder is a description of an algorithm, expressed in a subset of C++, optionally enhanced with fixed-point classes that can be provided by A|RT Library or SystemC [1].

A|RT Builder interprets C sequential statements, except function calls, as behavior and maps them sequentially into a HDL (Verilog or VHDL) process. Function calls result in structure by mapping them in HDL modules. These modules’ interfaces are defined by the C function parameters. By applying A|RT Builder’s concept a single execution of the C algorithm will always correspond to the execution of one clock cycle in the output HDL. Hence resources are not shared over clock cycles. Designers must use a specific coding style to describe sequential logic as required for FSMs in order to minimize area and schedule operations over clock cycles. Since A|RT Builder’s input must be *timed* C++, as explained in Section 2.3.1. This means that the HW architecture is reflected in the source. Changing or selecting architectures as enabled by Catapult-C’s “micro architecture what-if analysis” requires rewriting the A|RT Builder source. The integration of interfaces is also not supported. Typically A|RT Builder created HDL must be wrapped by hand-coded “glue” HDL. More details on A|RT Builder are provided in Section 4.4.1 and in the user manual [1].

In fact A|RT Builder requires C++ RTL modeling, since the output HDL is mapped “one to one.” This requires designers to be familiar with latency, pipelining, resource sharing etc., annihilating the benefits of a C

[5] A|RT Builder was sold to ARM in 2003.

based design flow, since no additional abstraction is gained.

Co-simulation is the simulation of refined, implementation specific prototype design units along with their formal (original, behavioral) counterparts. Those prototypes intended for HW implementation are generated for FPGAs, as in this thesis' scope. From the various existing FPGA prototyping methodologies I chose behavioral C to RTL synthesis.

Chapter 3

Design Environment

This chapter introduces the HW platform comprising the FPGA, the tool for block based (co-)simulations and other major design tools.

3.1 Hardware Platform

The prototyping platform is the RTS-DSP board from ARC Seibersdorf research. It contains a DSP part (including the target FPGA) for executing simulation algorithms and a CPU part handling peripheral tasks like communication to external systems. Figure 3.1 gives an overview of the board architecture. Detailed descriptions of the board are provided in [20, 13].

3.1.1 DSP part

This part contains a TMS320C6416 DSP from Texas Instruments (TI), clocked at 600 MHz, which is suitable for complex computations and algorithms. The second processing element is a Xilinx VirtexII XC2V2000 FPGA, whereas an auxiliary feature is the connection of the DSP and the CPU via dual ported BRAMs. The next section provides more details on the FPGA. Data transmission between DSP and FPGA is carried out by the EMIFB (external memory interface B), which is clocked at 100 MHz. EMIFB also connects four MByte of flash memory to the DSP, which contains the DSP executable and the FPGA bitstream file. EMIFA connects 32 MBytes of synchronous dynamic RAM (SDRAM) to the DSP. If necessary the system can be upgraded with up to 512 MBytes of commercial SODIMM (small outline dual in-line memory module) SDRAM [14].

3.1.2 FPGA

The dedicated dual ported RAM, mentioned in the previous section, handles the communication between the CPU and the DSP (Figure 3.1). In general, the remaining chip area is used for HW implementable, computational tasks

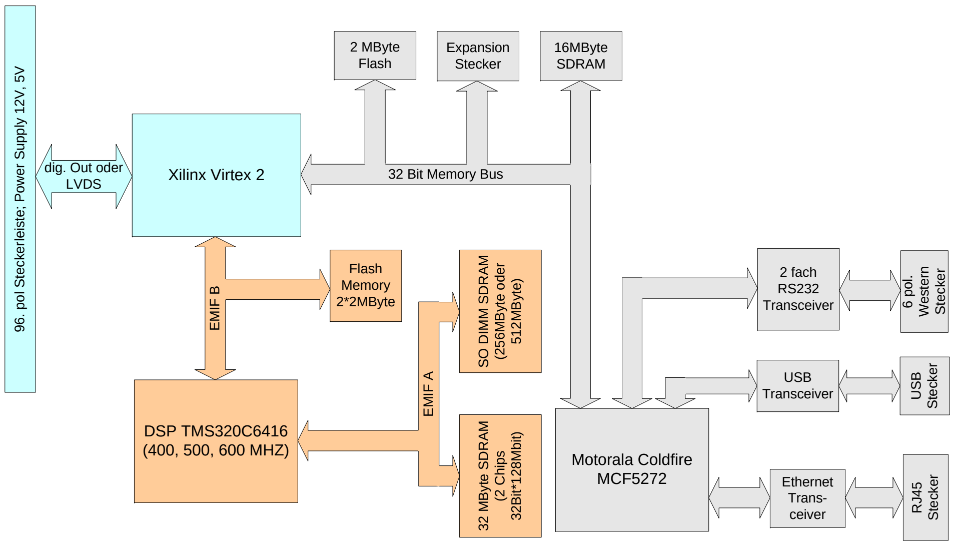


Figure 3.1: Block diagram of the RTS-DSP board.

which decrease the work load of the DSP. When the RTS-DSP board is used in conjunction with Rapid ProtoTyper (RPT) it carries the FPGA prototype, synthesized from a VHDL design. The already existing outermost VHDL shell, developed by ARCS during former projects, incorporates the prototype. This shell provides the data bus, address bus and all control signals, e.g., read/write, chip enable, etc. Furthermore, it was adopted for rapid prototyping by providing a particular clock solely for the prototype. This prototyping clock is derived from the 66 MHz CPU clock and might be multiplied or divided by means of the on-chip phase-locked loop (PLL), according to the requirements of certain prototypes, which allows for freedom in prototyped algorithms.

The FPGA features 2M system gates, 56 18-bit multipliers, 1008 kbits of BRAM and eight digital clock manager (DCMs). Interrupt pins of both DSP and CPU can be accessed by the FPGA, which allows for efficient data transmission to adjacent cores. E.g., the external interrupt pin of the CPU is used to signalize that the FPGA has completed its computations and output data is ready for carriage.

3.1.3 CPU Part

The CPU of the board is a Motorola Coldfire MCF5272 chip. Its main task is the data transmission between the FPGA and the CPU and the execution of the TCP/IP stack for the Ethernet IF. It is clocked at 66 MHz and features an integrated fast Ethernet controller (FEC), which controls a fast Ethernet transceiver LXT971A from Intel, carrying out the ISO-OSI (open system interconnection) physical layer. The EMIF of the CPU connects 16 MBytes of SDRAM, 4 MBytes of flash and the FPGA [14, 10].

3.1.4 Interfaces

The most important IFs for rapid prototyping are:

Ethernet connector Enables data transmission to the host PC. This port, whose data transfer rate is substantially lower than the low voltage differential signaling (LVDS) IF's data rate, was initially intended to transfer visualization data only.[1] Its data rate is sufficient for this purpose. However, when the board is used for rapid prototyping, the Ethernet port carries the main data stream, which might turn out to be the bottleneck in the data transmission path. However, small execution time of the simulation is not a primary goal of this thesis.

LVDS IF A high speed serial IF which enables transmission of external data or interconnection of two or more RTS-DSP boards.

JTAG (Joint Test Action Group) IFs enable direct programming of the FPGA and flash memories. An RS-232 (Recommended Standard-232) IF allows debugging of the CPU [14].

Once a new bitstream has to be loaded onto the FPGA it can be loaded via Ethernet, CPU and FPGA, directly into the DSP SDRAM (see Figure 3.1), wherefrom the DSP reconfigures the FPGA at runtime, i.e. no board reset is required.

3.2 Simulink

Verifying prototypes by co-simulation is crucial in rapid prototyping, as mentioned in Section 2.1. Simulink, a well known simulator in engineering, is used for this co-simulations.

Simulink is a software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. The scope of this work includes discrete systems only, i.e., systems with sampled time. Systems can

[1]The ARCS RTS-DSP board was designed for a (mobile communications) channel simulator, for more information refer to [14].

also be multirate, i.e., have different parts that are sampled or updated at different rates.

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. Block libraries, such as the “Signal Processing Blockset,” offer blocks performing common tasks of their field, e.g. digital filtering. Simulation results can be viewed on scope blocks or saved to workspace variables. However, performing co-simulation with a proprietary HW platform requires the use of specific, user defined blocks. Their creation is enabled with so called S-Functions, explained in the next section.

3.2.1 S-Functions

An S-Functions allows the integration of program code into Simulink blocks. This offers great freedom because any programmable behavior or action can be implemented. S-functions can be written in MATLAB m-code, C, C++, Ada, or Fortran. I use C++ S-Functions since C++ offers great flexibility and existing ARCS libraries are written in C. C++ S-Functions are compiled into MATLAB executable (MEX)-files, which requires special compiler options. MATLAB comes with the `mex` utility, which sets all these options and calls a (customizable) C++ compiler. In this thesis Microsoft’s Visual C++ (VC++) compiler, introduced in Section 3.3.2, is used.

Understanding how S-Functions work requires understanding how Simulink works. A Simulink block consists of a set of inputs, a set of states, and a set of outputs, whereas the outputs are a function of the sample time, the inputs, and the block’s states. Basically the execution of a discrete Simulink model proceeds in three stages:

- Initialize model.
- Calculate outputs.
- Update states.

First comes the model initialization where Simulink incorporates library blocks into the model, propagates widths, data types, and sample times, evaluates block parameters, determines the block execution order, and allocates memory. Then Simulink enters a simulation loop, where each pass through the loop is referred to as a simulation step. During each simulation step, Simulink executes each of the model’s blocks in the order determined during initialization. For each block, Simulink invokes functions that calculate the block’s outputs and states for the current sample time. This continues until the simulation time has elapsed.

An S-Function must contain a set of functions called by the Simulink scheduler. This set of callbacks and their execution order is depicted in Figure 3.2, whereas the `mdlUpdate()` callback may be omitted if not

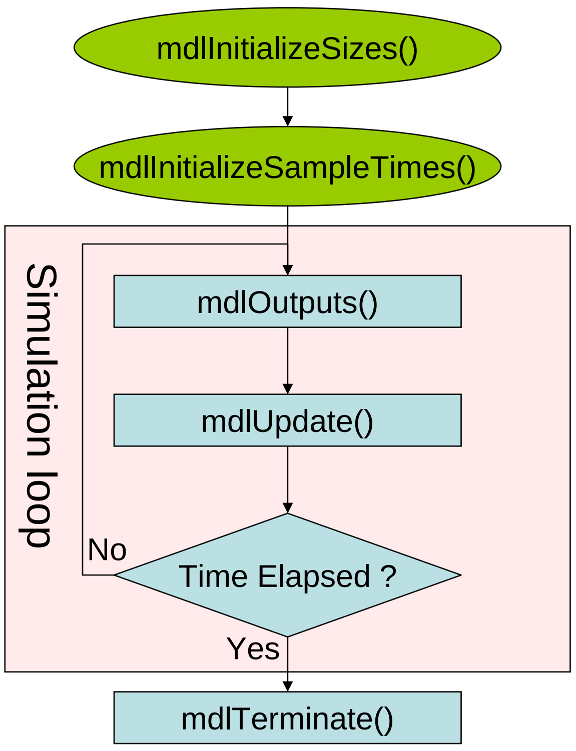


Figure 3.2: S-Function callbacks and their execution order.

used. There exist much more optional callbacks, which are exactly explained in [29]. Most important is the `mdlOutputs()` callback. As shown in Figure 3.2 it is invoked in each simulation step. Hence, all action for a blocks regular behavior (not initialization) is contained here. Macros, provided by the S-Function application programming IF (API) are used to obtain and store Simulink block information, e.g., `ssSetNumDiscStates()` or `ssSetSampleTime()`. All callbacks with "initialize" in their name are called once at simulation start-up in order to set initial values and define data sizes etc. `mdlTerminate()` is usually used to clean up allocated memory, since it is invoked as the very last function.

Simulink is used to co-simulate prototypes, integrated into Simulink by means of S-Functions, allowing to incorporate user defined functionality with callback functions.

3.3 Design Flow Tools

Creating a rapid prototyping environment requires the use of tools in a design flow. This section introduces third party tools whose usage was fixed prior to creation of the prototyping design flow. Hence they are considered to be part of the design environment.

3.3.1 Xilinx ISE

Xilinx' Integrated Software Environment (ISE) is an FPGA design software suite. It allows to net FPGA resources by creating a configuration for its interconnect matrix such that it behaves exactly as the design entry specification.

ISE enables to specify HW designs by HDL, netlists, intellectual property (IP)-cores etc. Such a design entry specification can be implemented, resulting in a bitstream for FPGA configuration. ISE comes with a set of tools performing parts of the entire design creation process [31]. Processes for bitstream creation are listed in the following:

1. **Synthesize:** Design entry files, viz. VHDL files in this thesis, are synthesized by the Xilinx Synthesis Tool. Synthesis allocates HW resources that behave in the same way as the corresponding language construct. Hence synthesizable HDL code is restricted in that it must conform to specific rules, since not all possible constructs are feasible in HW. This coding style is referred to as register transfer level (RTL) style. Synthesis tools output netlists for further processing by the following tools. These netlists contain information on used resources and their interconnection. Macro inference allows to allocate FPGA resources without instantiating them in the code, i.e., by following Xilinx coding styles synthesis results can be optimized.

2. **Implementation:** This process consumes, unlike the two others, a huge amount of computational power and, hence, time. It comprises three sub-processes:

   Translate: The translate process merges all input netlists and design constraint information in order to output a Xilinx Native Generic Database file, which describes the logical design reduced to Xilinx primitives.

   Map: The translated design is mapped into FPGA elements, such as configurable logic blocks (CLBs) and I/O buffers (IOBs). The output is a native circuit description file that physically represents the design mapped to the components in the Xilinx FPGA.

   Place and Route: This process takes a mapped design, places and routes it, and produces an Native Circuit Description file that is used as input for bitstream generation.

3. **Programming File Generation:** Eventually bitstreams for Xilinx device configuration are generated. These bitstreams must be downloaded to the FPGA in order to execute the desired function.

All listed processes have corresponding tools which have to be invoked consecutively for creation of a bitstream from a VHDL source, as required for rapid prototyping.

3.3.2 Visual C++

Microsoft's VC++ is part of the quite popular Visual Studio (VS) SW development platform on Windows. It provides a C++ compiler and linker, used for the rapid prototyping environment.

Typically Visual Studio is used with its GUI. Programmers type their code into an embedded editor and initiate compile and build (also known as link) steps by pressing buttons, which, in turn, call the corresponding command line tools.

Compiler: Prior to compilation comes a preprocessing stage, executing all preprocessor instructions and removing them from the file. The actual compiler analyzes the remaining C++ code and outputs object files, whose generation can be controlled by a variety of options, such as "inline function expansion." Unfortunately is this compiler not 100% compatible to ISO C++, e.g., partial template specialization is not supported.

Linker: Object files from the compiler are read and executable code is created. As well as compiling linking is customizable by options, e.g., "Output file name."

Automatic rapid prototyping requires the compiler and the linker to be accessible from an external program, which is possible since both provide a command line IF.

The RTS-DSP board from ARCS contains the Xilinx Virtex II FPGA for rapid prototyping. It communicates via Ethernet with Simulink's S-Functions, used for co-simulating prototypes. Xilinx' ISE is used for FPGA implementation and Microsoft's VC++ compiler creates the S-Function MEX files.

Chapter 4

Rapid Prototyping Methodology

Based upon theory introduced in Chapter 2, this chapter provides key concepts and fundamentals developed in order to implement the rapid prototyping tool: Rapid ProtoTyper (RPT). The simulator's role will be explained in detail, followed by methods used to integrate prototypes in co-simulation. These methods include analysis of the single input source, leading to creation of the FPGA prototype and required simulation software.

4.1 Design Flow Overview

What must be done in order to obtain prototyping blocks? A general, simple design flow, assembled from fundamental blocks, answers this question, while the detailed, implemented design flow is illustrated in Chapter 5.

The overall goal, as explained in Section 2.1 in detail, is replacing a Simulink block with its prototype. Further was defined, that this prototype has to be described in a C like programming language, which forms my system's input as illustrated in Figure 4.1 in a high-level, detail-omitting view. Following the "one source" paradigm in the five ones approach, mentioned in [27], the entire rapid prototyping flow is set up on a single source.[1] Each block in the diagram depicts the prototype at a different implementation stage, apart from the `Hardware in the Loop`[2] block being just a shell for another prototype realization. Since all blocks are data representations there is no information in this diagram on *how* data is compiled or generated. However, some third party tools with major influence on my concepts are introduced during discussion while the complete tool set is explained in Chapter 5. Attention should be paid to the arrow color since red illustrates

[1]Per definition of source code, this does not include any auto-generated code, as e.g. VHDL code (in our design flow).

[2]Convention in this chapter: typewriter font **refers to Figure 4.1**.

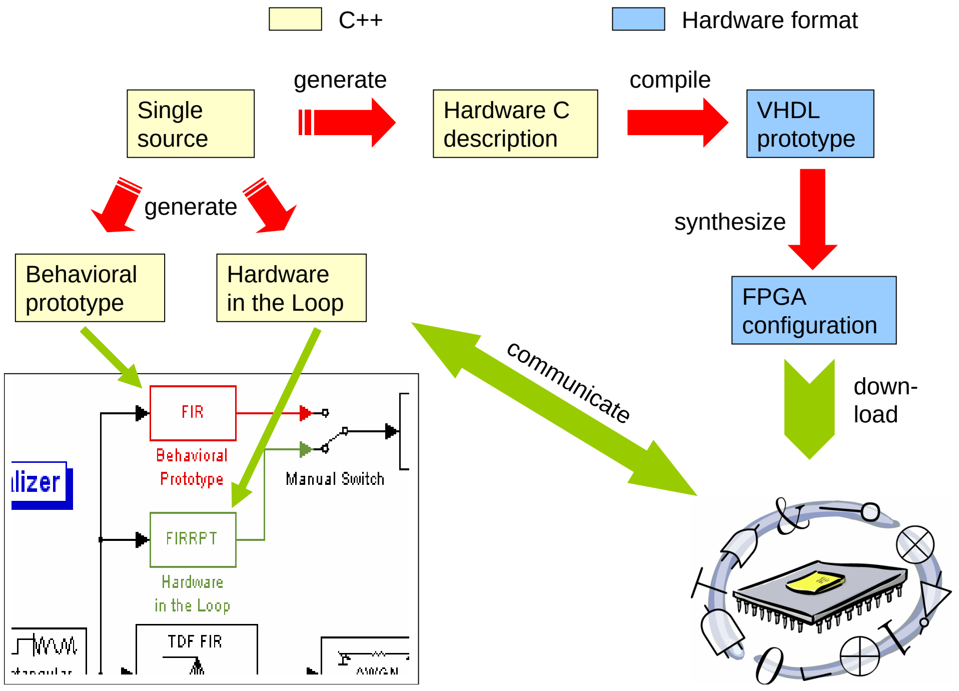


Figure 4.1: Simple, high-level design flow.

transformation steps among data representations opposed to green, indicating data *transfer*. Keeping track of the flow is alleviated by splitting it up in two branches:

1. Simulink generation.
2. FPGA generation.

Although partition by language, as provided in Figure 4.1, suggests togetherness of blocks with equal color the `Hardware C description` counts towards branch item 2 FPGA generation.

4.1.1 Simulink Generation

Co-simulation in Simulink requires the `Behavioral prototype` block and the `Hardware in the Loop` block to be implemented in terms of C++ S-Functions (for more information on S-Functions refer to Section 3.2.1). The `Behavioral prototype` block is solely PC-executed and therefore can be immediately generated from the `Single source`. This source, based upon C++, is unique in that all other prototype descriptions are generated from it and thus no further (prototype) programming is required. However, this approach is not new, e.g. Rupp also recommends the use of one source in [27]. As Figure 4.1 shows, the `Hardware in the Loop` block consequentially communicates with the FPGA suggesting that some interfacing concept is involved. Since this IF is also generated from the single source, which includes all this IF information (in detail explained in Section 4.2.1), the `Hardware in the Loop` block can be generated immediately, too.

4.1.2 FPGA Generation

Being RPT's base, C to VHDL conversion is illustrated in the upper right corner of Figure 4.1. Since writing an own C to VHDL conversion tool would have been too much effort I decided on using an external tool. Among the various alternatives discussed in Section 2.3.2, I chose A|RT Builder for reasons discussed in detail in Section 4.4.1. Since A|RT Builder requires input extended with specific code constructs, a `Hardware C description` must be generated to fit the tool's syntax. Once this description has been compiled, the resulting, HW synthesis compliant `VHDL prototype` models the prototype at RTL level. Finally this VHDL prototype is synthesized by Xilinx ISE and the resulting net lists are implemented in an `FPGA configuration` bitstream. In order to run the prototype on the FPGA it has to be configured with the bitstream. Finally communication between simulation and FPGA, via Ethernet, can be established.

4.2 Input Source

In order to describe a systems behavior as well as its IF a suitable input format is chosen. Further, its composition is analyzed and strategies for analysis and information extraction are presented.

Requirements: Since an entire simulation block is described by a single source, all information essential to the block must be contained in this source. To enable simulation-embedding the block's interface must be specified, including parameters such as the number of input/ouput (I/O) ports, their data types and sample rates. A detailed system level view from Simulink at prototype integration is provided in Section 4.3. In addition to the inevitable IF it is desirable to equip the block with behavior and, moreover, with state.

4.2.1 Generic-C

Generic-C, defined in COSSAP[3] and well known in EDA industry, was selected as prototyping language, since it comes with features allowing to tackle the above mentioned tasks with relative low effort. Generic-C is a C dialect with proprietary *language extensions*, designed to describe block

[3]Communications System Simulation Analysis Package.

```
#INCLUDE "fxp.h"
// data rate
#define R 16
// correlation signal depth
#define DEPTH 40

// input for correlation
INPUT_PORT(1) Fix<8,6> * u;
// output of correlation
OUTPUT_PORT(1) Fix<8,6> * y;

RATE(INPUT_PORT(1))=R;
RATE(OUTPUT_PORT(1))=R;

STATE Fix<8,6> x [DEPTH];
```

Figure 4.2: Generic-C header example.

based DSP systems, which made it a solution tailored for rapid prototyping. Fortunately the language extensions are mainly used to describe block IFs and affect system behavior code only marginally as depicted in the following.

A Generic-C description consists of a header section and two functions modeling a block's behavior.

Header Section: Each statement (not preprocessor instructions) in this section must begin with a Generic-C keyword as exemplified in Figure 4.2. An overview of Generic-C keywords and their associated declaration's enhanced Backus Naur form (EBNF) grammar is provided in the following:

`INPUT_PORT` declares an input port. The exact syntax for the input port statement is:

`"INPUT_PORT" "("` *index* `")"` *type* `"*"` *identifier* `";"`

Index must be a positive integer, *type* a C/C++ base type or fixed-point type from the A|RT library and *identifier* a valid C/C++ variable name.

`OUTPUT_PORT` declares an output port, whereas syntax rules are equivalent to the `INPUT_PORT` statement:

`"OUTPUT_PORT" "("` *index* `")"` *type* `"*"` *identifier* `";"`

`RATE` marks the beginning of a port's data rate declaration with the following syntax:

`"RATE" "(" "INPUT_PORT" | "OUTPUT_PORT" "("` *index* `")" ")" "="` *rate* `";"`

The declaration of the associated input port respectively output port, selected with *index*, must occur prior to the rate declaration. The *rate* itself can be any positive integer, but optimal HW results are achieved by using powers of two.

`STATE` is used to model a blocks state. Although variables, containing state information, can also be declared in the behavioral part, using the `STATE` construct is advantageous since tools can handle it explicitly. The `STATE` declaration, unlike all other Generic-C declarations, must not appear more than once per source, according to [26]. Its syntax rules are:

`"STATE"` *type identifier* `"["` *size* `"]" ";"`

For *type* and *identifier* apply the same rules as mentioned in the `INPUT_PORT` declaration. According to syntax rules, states *must always* be declared as arrays.

RPT does not support all Generic-C constructs since this would have been too much implementation effort. However, those not supported, e.g. `PARAMETER` and `RAM`, are not essential to describe a blocks behavior and, furthermore, workarounds exist, which have the same effect.

Algorithmic Part: Unlike the header section this part contains only minor Generic-C specifics. System behavior is modeled by an initialization function and an algorithm function. The former does exactly what its name suggests; hence it is called only once at simulation begin. Usually solely the state variable, i.e. the variable declared by `STATE`, is initialized here. For a block named *Prototype* the Generic-C file name must be "*Prototype*.gc" and the initialization function's syntax is defined in the following:

`"init_`*Prototype*`" "(" ")" "{"` *body* `"}"`

Compared to regular function calls it is obvious that no return type and no parameters are allowed. Parameters are not required, since implicit access to the state variable is granted. The function *body* is, as usual in C, embedded in braces, must *not* contain any proprietary extensions and thus can be compiled by standard compliant C/C++ compilers.

The algorithm function actually contains the block's behavior. The syntax is defined as following in EBNF:

`"`*Prototype*`" "(" ")" "{"` *body* `"}"`

Since, equal to the initialization function, no parameters are allowed, the algorithm implicitly has access to IF data from the header section. Port variables, although declared as pointers, are contiguous arrays and thus may be accessed like the state variable with the index operator `[]`. Basically the body may be coded in whatever style designers prefer, although hardware creation deserves attention because the coding style influences resulting hardware architectures dramatically as discussed in Section 4.4.2.

Summarized, a Generic-C provides an IF section, containing I/O port and state information, and an algorithmic part, defining the blocks behavior.

4.2.2 Input Analysis

Rupp already wrote a tool performing Generic-C conversion, entitled GenC and accurately described in [26]. Various outputs can be generated, including A|RT Builder code and Simulink S-Functions. Maier used GenC in his diploma thesis [18] to generate S-Functions for rapid DSP-prototyping. In order to generate DSP code he wrote an add-on analyzing GenC's S-Functions, which is an interesting approach in that this add-on gets around analyzing Generic-C. Since Maier's prototyping system carries out the same task as mine, except that his target is a DSP, I considered applying his approach in my system. However, during some case studies it turned out that S-Functions are harder to analyze than Generic-C code since the required information is hidden in an accumulation of surrounding code. Additionally GenC generated S-Functions use improper, old-style memory management increasing block invocation time. Another problem were several Generic-C input issues (white-space sensitivity, sequence sensitivity, insufficient type support, faulty error handling . . .), which finally convinced me to set up a new tool chain including Generic-C analysis. For avoiding the problems mentioned above I decided to use a more general, object oriented approach. A scanner-parser tool, using token streams, extracts pure information from Generic-C sources and provides it in a uniform, clean representation. Relating to Figure 4.1 this representation replaces the `Generic-C source` and forms the basis, i.e.the input, for generator tools.

In order to create token streams from Generic-C a scanner is used. It reads the input file and extracts command strings, terminated by semicolons. While scanning these strings white spaces are discarded, additionally to C++ line comments `//` and section comments `/* ...*/`. Resulting tokens are analyzed by a parser creating a database, particularly from Generic-C IF description constructs as mentioned in Section 4.2.1. This database contains all relevant information from the Generic-C source like I/O port numbers, types, names, rates and state information. Additional to ANSI C types fixed-point types are supported, defined in the A|RT library [2]. By means of these types, variables of arbitrary length up to 32 bits are possible, whereas the binary point may be specified at any position in the mantissa.

Generic-C function contents are not analyzed by the parser but saved in the database file together with the prototype IF information mentioned above.

Summarized, Generic-C is used as the input source language. It contains a proprietary header section, used to indicate IF information as ports, and functions for initialization and algorithmic behavior. Generic-C files are processed by a scanner-parser tool providing a uniform database.

4.3 Simulation Semantics

Considering Simulink model creation, prototype integration is a core issue. Prototype requirements and attributes of Simulink, viewed from simulation data flow perspective, are discussed.

First of all, before any code can be generated, a simulation semantics has to be established. An eminent feature of Simulink is its time based simulation engine. It supports continuous time systems as well as discrete time systems. Since my rapid prototyping system focuses on digital hardware only I do not have to deal with Simulink's continuous time features. For discrete time systems an appropriate solver exists, which guarantees the following:

- Simulation time advances in fundamental *sample time* steps, whereas each sample time (in multi rate systems) must be an integral multiple of the fundamental sample time.
- Each block's algorithm is only invoked once per sample time hit, contrary to continuous time systems.
- Prior to block invocation block inputs are updated, analogous block outputs at the invocation end.

When using Simulink for prototype co-simulation some issues have to be considered, as explained in the following.

4.3.1 Background

In addition to supporting single rate systems RPT must also support multi rate systems. Simulink offers the *multitasking* scheduling algorithm for this purpose. All blocks with common sample times are assigned to a separate task, which is only invoked when a sample time hit occurs. In multi rate systems with sample times of adjacent blocks being integral multiples of each other this multitasking approach can lead to unexpected system behavior due to dependencies in the tasking sequence. More precise it may emerge that a block, viewed in the model's signal propagation flow, is executed *after* its successor when a sample time hit occurs at both blocks and they reside in different tasks.

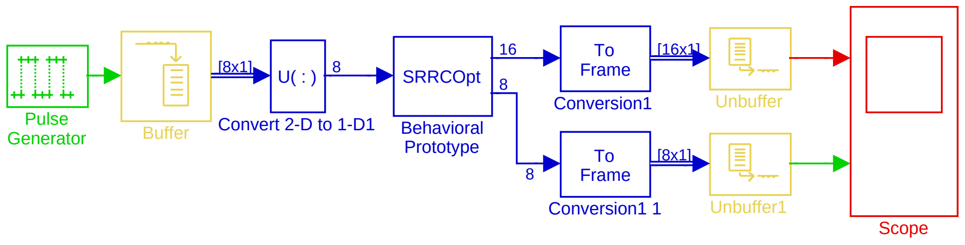


Figure 4.3: Multi rate block processing simulation.

Simulations with a high count of blocks (several hundred) and deep hierarchies can cause a severe performance decrease due to the overhead caused by the scheduling algorithm. When the simulation additionally is very exhaustive it is desirable to reduce the overall simulation time and subsequently the scheduling overhead. To tackle this task one may use block processing. Thus, in the simplest case, the algorithm is executed iteratively according to the block processing size, while the block itself is only invoked once by Simulink's scheduler. In other words the scheduler's multiple block invocations are shifted *inside* the block.

4.3.2 Approach With Generic-C

Since the source language is Generic-C, the above mentioned problems can be tackled by means of the language integrated features. Thus a data rate can be specified for each input and output port, which, according to my simulation semantics, corresponds to the block processing size. However, the data rate not only allows to solve the scheduling problem, it also addresses the multi rate problem. By specifying different data rates for ports one can create a multi rate block, though the block, in Simulink only uses one sample rate. Figure 4.3 exemplifies a typical multi rate simulation, whereas different sample times are indicated by associated colors. Designers, who code blocks in Generic-C, have full control over the handling of sample rates because it takes place in the Generic-C code. Each port is mapped into an array with a size given by the data rate. This array size corresponds to the signal dimensions displayed in Figure 4.3. Full control means that any value of the array can be accessed via the index operator `[]`. Semantically, the value provided by the array index corresponds to a port value at the time given by this index. Designers have to take care of processing the entire block in this case. It would have been possible to restrict their data access rights and perform the iteration over the array, including port indexing, automatically to exclude potential human errors. However, in a case study it turned out that this would only lead to satisfying results when all arrays are of equal size, i.e. in the case of a single rate system. To offer more flexibility I decided to let designers take care of port accessing. Section 6.2 provides Generic-C source code illustrating the actual port access methodology.

Eventually a Simulink data representation must be chosen, regarding criteria such as flexibility and compatibility. Simulink's block based data is usually represented by frames. In order to be compatible with the data representation of [18], I have chosen vectors, indicated in the diagram by numbers. By using Simulink's `Convert 2-D to 1-D` library block, frames can be converted, without information loss, to vectors and vice versa with the `To Frame` block. Besides sample time, Simulink block routing lines are characterized by their data type. Usually double precision floating point format is used. However, in DSP systems specialized formats, such as fixed-point, are often preferred. A|RT Builder comes with its own, proprietary fixed-point format, which can be integrated into Simulink via a user-defined routing data type. Unfortunately such a type is incompatible to library blocks like scopes, which is a substantial drawback. Hence double precision floating point format is used whereupon Generic-C data types are used only inside prototype blocks (S-Functions).

Summarized, multi rate support as well as simulation scheduling effort attenuation is achieved by block processing and Simulink routing is represented in double precision floating point format.

4.4 Hardware Generation

Branch two in Figure 4.1, introduced in Section 4.1, illustrates stages of FPGA prototype creation. Based on these several stages are concretized and refined, more precisely the following is established:

- Actual C to VHDL conversion methodology.
- FPGA embedding of the Prototype.
- Prototype access and control.

Before a prototype can be implemented on FPGAs, it must exist as HDL module written in RTL style, such that it is possible to apply hardware synthesis with common tools as, e.g. Mentor Graphics' Leonardo Spectrum [22]. To maintain automation in RPT the prototype must be *generated* from Generic-C code, unlike manual porting in [30]. Ideally, the prototype should exist as entity-architecture pair to ease integration into other VHDL code. Fortunately, Generic-C language constructs, in particular `INPUT_PORT` and `OUTPUT_PORT`, allow for direct mapping into a VHDL entity.

4.4.1 A|RT Builder

Since compiling an *untimed* C/C++ algorithm[4] into synthesis compliant VHDL is very difficult, as pointed out in Section 2.3, designing such a tool

[4]This does not apply for SystemC which is timed and, amongst others, designed for HW synthesis; the terms timed and untimed are explained in Section 2.3.

```
void Prototype (
  const Int<4> a,
  const Int<4> b,
  short & c)
{
   #pragma OUT c
}
```

```
library ieee;
use ieee.std_logic_1164.all;
use work.artbuilderpack_numeric.all;
use ieee.numeric_std.all;

entity Prototype is
  port (
    a: in std_logic_vector(3 downto 0);
    b: in std_logic_vector(3 downto 0);
    c: out std_logic_vector(15 downto 0)
  );
end Prototype;
```

Figure 4.4: C function and corresponding VHDL entity.

would be enough work for a PhD thesis. Hence I selected the third party tool A|RT Builder, based upon the following reasons:

- Unfortunately Catapult-C was not available to me.
- A|RT Builder comes with a fixed-point library implemented both in C++ and synthesizable VHDL.
- A|RT Builder does not require its input to be modified with proprietary language constructs (except compiler pragmas).
- A|RT Builder was available to me from the first day of my diploma thesis.

Unfortunately A|RT Builder is no longer commercially available, since Adelante Technologies sold its A|RT technology to ARM, which accordingly prohibits commercial use of RPT. Nevertheless A|RT Builder is suitable for "proof of concept."

A|RT Builder Hardware Semantics

A|RT Builder's hardware semantics is described in detail in [1] while here a brief overview about key concepts is provided.

In general, A|RT Builder creates HW as parallel as possible. For each C++ operation an equivalent RTL VHDL operation is generated, which is, in other words, a "one to one" mapping. A VHDL component is generated from a C function whereas its parameter list is mapped into input and output ports, as indicated in Figure 4.4. In particular outputs in the parameter list have to be indicated to A|RT Builder with the `OUT` pragma. The template type `Int<4>` is one of the fixed-point types from the A|RT Library, mentioned in Section 4.2.2. As shown by the bit widths in the VHDL entity, A|RT Builder supports these types and, furthermore, A|RT Builder supports corresponding HW arithmetics. Calls of other C functions from the outermost function result in instantiated VHDL components and, thus, a function hierarchy gives a VHDL component hierarchy or, in other words, a facility to describe structure.

Combinational Logic

Combinational logic is generated from C operations like addition, `if` statements or `for` loops. The function in Figure 4.4 could be extended with some functionality as exemplified in the left of Figure 4.5. To its right the resulting architecture is shown, with only a purely combinational process containing the C function's operations in data-dependency correct sequence.[5] I.e., for instance variables for intermediate results may be removed for optimization, however, the sequence of operations is preserved. As Figure 4.5 shows not only the operation sequence but also operations themselves are preserved. Hence, e.g., the C `if` becomes a VHDL `if`, the C | becomes a VHDL `or` and also the `for` loop is maintained. For satisfying VHDL syntax requirements, being tighter than in C, A|RT Builder adds casting operations. Furthermore A|RT Builder sets VHDL variable bit-widths automatically for correct arithmetics as shown in the `COMPUTE_PROC` process' declaration section in Figure 4.5. E.g., `n2` is a 20 bit variable (`19 downto 0`) since it gets assigned the product of a four bit and a 16 bit variable (refer to the `for` loop's body).

According HW synthesis rules this VHDL description will become a purely combinational circuit because no clock is involved and the output is solely a function of the inputs. Synthesis tools will infer HW operators for VHDL operators, a multiplexer for the if and the for loop will be unrolled into separate multiply accumulates (MACs), i.e. each operation is mapped into an associated HW unit, which is fully parallel implementation. So far outputs at a specific time are a pure function of the inputs. However often it would be useful to equip a design with state, i.e. to remember values. This is required to realize particular behavior like counting and, furthermore, when area limitations force resource sharing or a design should be pipelined, i.e. its long critical paths are split up by register insertion.

Sequential Logic

When state is demanded a notion of time must be introduced. One execution of the top-level C function will consume one set of inputs and produce one set of outputs. This single execution is called a time frame, corresponding to one clock cycle. Since outputs are a function of both inputs and the present state, A|RT Builder's generated architecture can be imagined as a Mealy machine, illustrated in Figure 4.6. When values in a C function are preserved beyond one time frame, i.e. one execution of the function, state is modeled. Common techniques are the use of variables with a `static`

[5] `cast()` is an A|RT Builder specific type conversion.

```
#include <fxp.h>

void Prototype (
  const Int<4> a,
  const Int<4> b,
  short & c)
{
   #pragma OUT c

   Int<4> d = a + b;
   Int<4> e = d + Int<4>(1);
   Int<4> f = e | d;
   if (f > e)
      c = f - e;
   else
      c = e - f;
   for (short i = 0; i<4; ++i)
      c += a*i;
}
```

```
architecture rtl of Prototype is begin
  COMPUTE_PROC: process(a, b)
    variable a_n1:signed(3 downto 0);
    variable b_n1:signed(3 downto 0);
    variable c_n1:signed(15 downto 0);
    variable d: signed(3 downto 0);
    variable e: signed(3 downto 0);
    variable f: signed(3 downto 0);
    variable n1: boolean;
    variable n2: signed(19 downto 0);
    variable n3: signed(15 downto 0);
    variable n4: signed(15 downto 0);
    variable n5: signed(15 downto 0);
    variable n6: signed(15 downto 0);
    variable n7: signed(15 downto 0);
  begin
    a_n1 := signed(a);
    b_n1 := signed(b);
    d := a_n1 + b_n1;
    e := d + signed'("0001");
    f := e or d;
    n1 := f > e;
    if (n1) then
       n6 := cast(f,12,0);
       n7 := cast(e,12,0);
       c_n1 := n6 - n7;
    else
       n5 := cast(f,12,0);
       n4 := cast(e,12,0);
       c_n1 := n4 - n5;
    end if;
    for i in 0 to 3 loop
      n2 := a_n1 * to_signed(i,16);
       n3 := cast(n2,-4,0);
        c_n1 := c_n1 + n3;
    end loop;
    c <= std_logic_vector(c_n1);
 end process;
end rtl;
```

Figure 4.5: C function and corresponding VHDL architecture.

storage class specifier or those declared in a global scope. They typically preserve their value(s) for the entire program execution time and they are initialized all together at program start — behavior similar to hardware state. When A|RT Builder finds one of these constructs it maps them into synthesizable VHDL register code in a sequential process as illustrated in Figure 4.7. This process updates its signals, i.e. the system's state, only at clock edges or changes of the reset signal, as common in synchronous design. The two state signals, `q_gc_Statexx_r` and `index_r`, are modeled in C as global fixed-point array and static fixed-point integer, respectively. Clock and reset are signals which do not occur in the C code, thus A|RT Builder

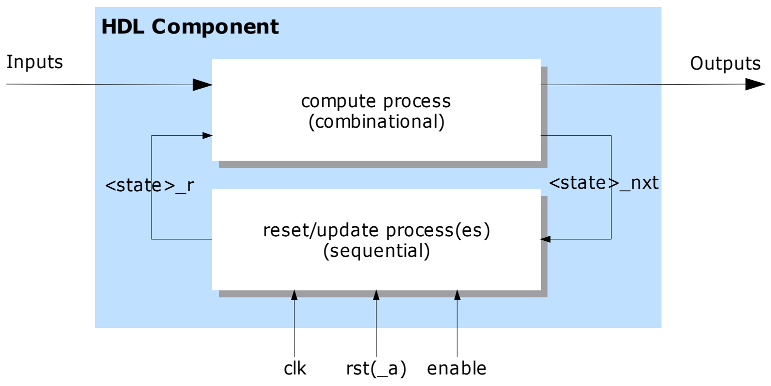

HDL Component
Inputs
Outputs
compute process (combinational)
<state>_r
<state>_nxt
reset/update process(es) (sequential)
clk
rst(_a)
enable

Figure 4.6: A|RT Builder's HDL component architecture.

```
RESET_UPDATE_PROC: process(clk,rst_a)
   variable q_gc_Statexx: signed_15d0_array_6d0; -- (state)
   variable index: unsigned(7 downto 0); -- (state)

begin
   if (rst_a = '1') then
      for i_n2 in 0 to 6 loop
         q_gc_Statexx(i_n2) := signed'("0000000000000000");
      end loop;
      index := unsigned'("00000000");
            -- copy local variables to next value for state
      for d0 in 6 downto 0 loop
         q_gc_Statexx_r(d0) <= std_logic_vector(q_gc_Statexx(d0));
      end loop;
      index_r <= std_logic_vector(index);
   else
      -- update state at clock transition only
      if (clk'event AND clk = '1') then
         if (enable = '1') then
            q_gc_Statexx_r <= q_gc_Statexx_nxt;
            index_r <= index_nxt;
         end if;
      end if;
   end if;
end process;
```

Figure 4.7: A|RT Builder-generated sequential process.

adds them automatically to the VHDL entity when it finds state in C. Often it is desired to prevent state updates for some clock cycles, e.g. when input data has to arrive. This behavior is achieved by usage of flip-flops with enable input, also automatically added to the entity by A|RT Builder.

Furthermore A|RT Builder supports advanced HW design methodologies such as pipelining or clock gating, realized by obeying particular coding styles or by using A|RT Builder `#pragma` instructions, respectively. Detailed descriptions are available in [1].

A|RT Builder has been selected to perform C to VHDL conversion in the design flow. Key concepts of HW generation, including combinatorial and sequential HW, are explained in detail enabling to understand later design examples.

4.4.2 Coding Style

Since A|RT Builder cannot support all C++ constructs, as while loops or sophisticated pointer usage, only a subset of C++ is allowed. When designers code Generic-C prototypes this must be regarded, because the Generic-C function bodies are mapped directly into the A|RT Builder source code. When describing an algorithm in a C function, designers usually write the function such, that one invocation produces the desired output. Thus, according to A|RT Builder's time frame concept, as explained in Section 4.4.1, all functionality had to take place in one clock cycle, consuming vast amounts of HW resources. Furthermore, since Generic-C data rates are realized by block processing, as depicted in Section 4.3, and an entire data block with size N is processed in one block invocation, the HW is instantiated N times.

In order to fit prototypes into the FPGA a resource sharing methodology is used. Multiple invocations of the algorithm share the same HW, thus producing the output step by step. How many clock cycles, or time frames in A|RT Builder terminology, are used is the designer's choice. They just have to set the system flag `done` in the last cycle to indicate algorithm termination as demonstrated in Figure 4.8. In order to preserve algorithm advances they must be stored in a state variable, which typically is a counter like `index`. Algorithm termination has to be tested in each invocation and, once it is true, the counter must be reset. The algorithm itself may be described by a construct like

```
y[index] = DoIt(u[index]);
```

The prototypes functionality is contained in `DoIt()` which is shared over all inputs `u[index]`. No matter how many times the algorithm is invoked `DoIt()`-HW resources are allocated only once, while different inputs are applied by means of the indexing operator. In general, indexed read operations, like `u[index]`, result in HW in multiplexors (MUXs), while address decoders are inferred for indexed write operations.

```
    static Uint<8> index = 0;

⋮

    ++index;
    if (index == N) // N is the data rate
    {
        index = 0;
        done = true;
    }
}
```

Figure 4.8: Generic-C coding style example.

Obviously the entire Generic-C code is styled for A|RT Builder but nevertheless it must also be implementable as S-Function to obtain correct behavioral prototypes. By invoking the algorithm function iteratively until `done` is set, behavioral results are equal to HW results. However state variables cause simulation errors, when their block is instantiated more than once. Since a static or global variable exists only once, no matter how often an S-Function block exists in one Simulink model, these blocks will influence mutually and thus falsify simulation results. To provide each block with a distinct set of state variables, S-Function dynamic link libraries (DLLs), referred to as MEX files, should be created separately for each block.

Generic-C prototypes must be described in a style which is mainly determined by A|RT Builder. Algorithm functions are invoked iteratively until termination is indicated by a system flag.

4.4.3 Prototype Embedding

Once the VHDL prototype has been generated it must be possible to access it from the "outside world" for supplying it with simulation data. The "outside world" constitutes an already existing outermost or top shell, provided by ARCS, for RPT. This top shell makes the CPU bus connected to the FPGA (see Section 3.1) available, which transmits simulation data. In order to offer simulation data for more than one bus cycle it must be stored in some kind of memory, constituting an interface between top shell and prototype. This IF, referred to as wrapper, encapsulates the prototype and provides a non generic, bus compliant IF on the top shell's part. Furthermore it should be possible to use an arbitrary number of ports, and thus an arbitrary number of memories. Hence, the wrapper is *generated.* Each prototype port can have its own data type, always being an assembly of bits, and its own data rate,

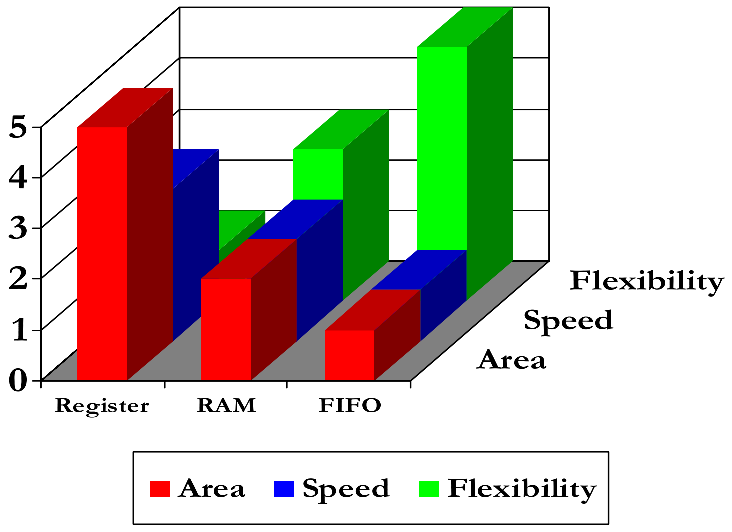


Figure 4.9: Wrapper performance in grades: 1-best, 5-worst.

which will result in different memory sizes. To keep the middleware[6] on the prototyping platform's CPU simple, all input port memories are mapped onto a contiguous memory map, correspondingly all output ports on their own memory map.

Wrapper Architectures

Three wrapper architectures, each with distinct attributes, have been identified. They are classified by their IF memory type as illustrated in Figure 4.9. Area and speed, two classical HW criteria, are compared along with flexibility. Flexibility means here, how much and what data is accessible in one clock cycle. The y-axis displays a performance estimation, in grades from one, being the best, to five, accordingly the worst. Diagram contents are supported in the following:

Register Wrapper: Most powerful of all wrappers, this one uses slice flip-flops as IF memory. Flip-flop memories are beneficial in terms of data accessibility, i.e. they *always* provide *all* data. A useful feature for highly parallel prototypes, requiring up to the entire memory in a single clock cycle. Hence the register wrapper is awarded a one in flexibility. Despite this perfect flexibility flip-flops are costly in terms of chip area, giving a five in area. Flip flops have their outputs

[6] μCLinux adapted for Motorola's Coldfire is used to handle communication and peripheral tasks [15], described in Section 5.2.2.

established "immediately" after their active clock edge, thus being very fast. Nevertheless, I awarded a three in speed for the register wrapper, since large MUX trees are required to pick out a word of a register file. Although delay times grow logarithmic only, they are substantial, even for small memories. Furthermore, the MUX effort supports the area grade.

RAM Wrapper: Another memory option are Xilinx specific block RAMs, existing in form of hard macros[7] on Virtex II FPGAs. Compared to register files they provide the advantage of integrated address decoding, saving MUX trees, although the address must still be computed by the prototype itself, which gives, in a whole, a two for area consumption. Block RAMs have a registered data output, which introduces a latency cycle, but eliminates combinatorial delay. Address-computation combinatorial delay included, speed is also awarded a two. Thus, in terms of area and speed, RAM wrappers are superior to register wrappers, but RAM wrappers have shortcomings in flexibility. Since RAMs cannot provide more than one data word per read access, highly parallel prototypes, requiring more than one word simultaneously, will lack efficiency when used along with RAM wrappers. The RAM must be read iteratively until all words are present, resulting in a three for flexibility.

Alternative to block RAMs Xilinx FPGAs offer distributed RAMs, assembled from configurable logic blocks (CLBs), the FPGA's general purpose resource. Unlike BRAMs distributed RAMs support arbitrary bit widths and furthermore these RAMs are read asynchronously, i.e. RAM outputs appear immediately when applying the address (after a combinatorial delay). However, I did not consider distributed RAMs in Figure 4.9. Distributed RAMs consume significantly more area than block RAMs, but offer, on the other hand, more flexibility.

FIFO Wrapper: FIFO memories are a substantially different from the previous types in that they do not allow random access. Thus a FIFO just allows read and write operations without an address, having severe consequences on Generic-C descriptions. Since data words are provided in a predetermined sequence (first in, first out), indexing operations, e.g., `u[index]`, do not make sense. Using just the port names themselves, e.g., `u`, is a reasonable solution. In addition each FIFO write access *enqueues* a new data word while each read access *removes* one, disallowing redundant accesses. Multirate systems, having FIFO sizes depending on data rates, are thus more difficult to code because the number of accesses must fit *exactly* to the size. In view

[7]Hard macros are "mini ASICs" integrated on the FPGA.

of these tight restrictions I awarded FIFO wrappers a five in flexibility. However, this is not such a severe drawback, since most streaming operations, in particular those with a single rate, do not require more flexibility.

Due to FIFO restrictions area and speed consuming logic can be saved. More precisely, since FIFOs compute their addresses internally no addressing logic is required, whereby the data path only includes the algorithm's operations. Regarding this along with efficient block RAM implementations of FIFOs, speed and area are issued a one.

Interfacing Concept

Simulation data is sent to the prototyping platform, and further to the FPGA, in a block bus operation via the CPU's 32 bit bus. Wrapper memories are filled in one continuous write operation, since they are mapped in a contiguous memory map. When the middleware finished data transmission to the input ports, the prototype is enabled by setting a control register in the FPGA which subsequently will begin computation. Depending on the number of iterations, scheduled in the Generic-C source, the prototype finishes computation after the equivalent amount of clock cycles and sets an interrupt flag, while putting itself into idle state. This interrupt flag is set by the `done` variable, mentioned in Section 4.4.2. The control register, used to enable FPGA computation and initially set by the middleware, is reset when the interrupt occurs. The board CPU's interrupt handler uploads the prototype's results via the CPU bus and Ethernet to the PC and eventually resets the interrupt flag on the FPGA.

Clocking

Prototypes can have, according to their C algorithm, very long critical paths. In real time systems, which operate at a constant data rate and thus a constant clock frequency, this is a problem when timing constraints cannot be met. It is possible to adjust the clock frequency to the critical path. To leave all other FPGA hardware unaffected a separate clock is used solely by the prototype. This prototyping clock is generated by an on-chip phase locked loop (PLL) in a Digital Clock Manager (DCM) unit and derived from the board CPU's clock. Since it is synchronous to the DCM source clock it is, in turn, synchronous to the board's CPU clock. This renders synchronization logic needless, however, care must be taken when using arbitrary, asynchronous clocks!

FPGA prototype creation was discussed in detail, including the C to VHDL conversion methodology, prototype wrapping and interfacing concepts.

4.5 Software Generation

In order to obtain a behavioral prototype it must be generated from the Generic-C source just as well as the FPGA prototype as illustrated in Figure 4.1. This section explains in detail what these S-Functions comprise.

"Simulink generation," the first branch in Figure 4.1, mentioned in Section 4.1, illustrates how Simulink blocks are generated. Two Simulink blocks, the `Behavioral prototype` and the `Hardware in the Loop` block, are generated from Generic-C. However, in Section 4.2.2 this high-level Generic-C generation concept has already been refined, resulting in a database as generator source. User defined Simulink blocks must be written as S-Functions, which are described in Section 3.2.1 in detail. I preferred to use C++ S-Functions, although Simulink supports Ada, Fortran and m code as well.

4.5.1 Encapsulation

The `Behavioral prototype`, which does not access external resources, can be generated from solely the database. The `Hardware in the Loop` block, in contrast, needs to access the FPGA, implying that additional information is required. An approach to create these S-Functions is to use a standard S-Function, e.g., the template provided by The MathWorks, and copy information extracted from the Generic-C source along with the standard S-Function callbacks directly into a file as used in [18]. This is a good solution when S-Functions are short and requirements do not change. However I decided to separate S-Function specific code from prototype functionality, providing easy reuse of prototype code, e.g. in a C++ test bench. Thus the prototype is encapsulated in a C++ module, consisting of header file and implementation. S-Functions include the header (by `#include`), instantiate a prototype object and access its functionality via class methods. Since S-Functions access prototype functionality from another file they can be written prototype independent, which allows to use always the same S-Function template with just its name adapted to the prototype. In order to handle arbitrary port counts, sizes and types these S-Function templates are written generically to be customized by the C++ preprocessor.

Prototypes, particularly the FPGA prototype, are accessed via software layers. Figure 4.10 illustrates this software layers, coarsely split into a PC section and a RTS-DSP board section. The left branch for the behavioral prototype is explained in the following section.

4.5.2 Behavioral Prototype Block

Behavioral prototypes are compiled from the behavioral S-Function template along with the prototype module, called `Behavioral prototype` and `Prototype module` in Figure 4.10 respectively. The latter contains a pro-

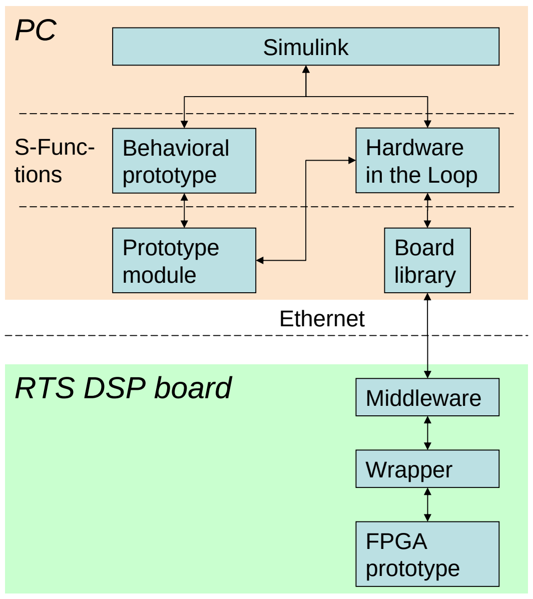


Figure 4.10: Data flow between Simulink and the prototype.

totype class, being the prototypes *behavioral* model. To adequately model prototype behavior this class offers two methods to S-Functions: an initialization method and an algorithm method, just as the Generic-C source itself. At simulation start-up Simulink invokes the initialization callback, which subsequently invokes the prototypes initialization method, placed in this callback. The algorithm method is placed in the output callback, which is invoked at sample time hits. The S-Function only provides input and output data pointers to the prototype. Port counts and sizes are set by the preprocessor, which replaces formal values with actual ones from the prototype module. Port types are set to double precision floating point format at this position, since the template S-Function is type independent and conversion to Generic-C port types takes place in the prototype module. As Section 4.4.2 describes, the algorithm is invoked iteratively to satisfy area criteria of FPGA prototypes. Since the Generic-C algorithm itself is copied into the prototype module "as is," these iterations must be also performed in SW, which is implemented by a loop, invoking the algorithm until the system variable `done` indicates to exit the loop.

4.5.3 Hardware In The Loop Block

Unlike the algorithm S-Function (behavioral model) the prototype S-Function does *not* execute behavioral code. Its task is solely accessing the FPGA prototype, as obvious in the right branch of Figure 4.10. The Simulink part of the `Hardware in the Loop` S-Function, describing input and output ports, is equal to the behavioral prototype S-Function. Although no algorithmic prototype functionality is used here, the prototype module is still required since it contains two functions which convert simulation data from double precision floating point format to binary fixed-point data and vice versa. Binary fixed-point format is required by the FPGA prototype. I decided to put all generated Simulink data into one module to get a moderate file count. The two data conversion functions are generated, since they have to use Generic-C port types and sizes. When all input port data has been converted to binary format, it is packed into one array. Multiple input ports are mapped into the array in a determined sequence, also known by the VHDL wrapper to enable data assignment to input port memories. Then the array is sent via Ethernet to the RTS-DSP board. A C++ library for the RTS-DSP board offers transmit, receive and control functions. After the prototype has received its input data a control command, issued in the S-Function, enables prototype computation by setting the corresponding FPGA flag. Once the prototype finished, all results are sent to the PC, converted back to double precision floating point format and finally made available to Simulink.

Summarized, all code intended for use in Simulink and generated from Generic-C is encapsulated in a prototype module, *specific* for each prototype. Simulink blocks are compiled from this module and generic S-Function templates. Board library functions allow to access the FPGA from Simulink.

4.6 Refined Design Flow

Based upon all insights and refinements of the generation concept, won in this chapter, the simple design flow in Figure 4.1 can be enhanced. Figure 4.11 illustrates the design flow including details required for implementation. However this design flow focusses only on data and their corresponding transformations. The final, implemented flow with all *tools* is introduced in Chapter 5.

Prototypes must be represented by a single source written in Generic-C. This source is used to generate a behavioral prototype S-Function, a HIL S-Function and its associated FPGA prototype.

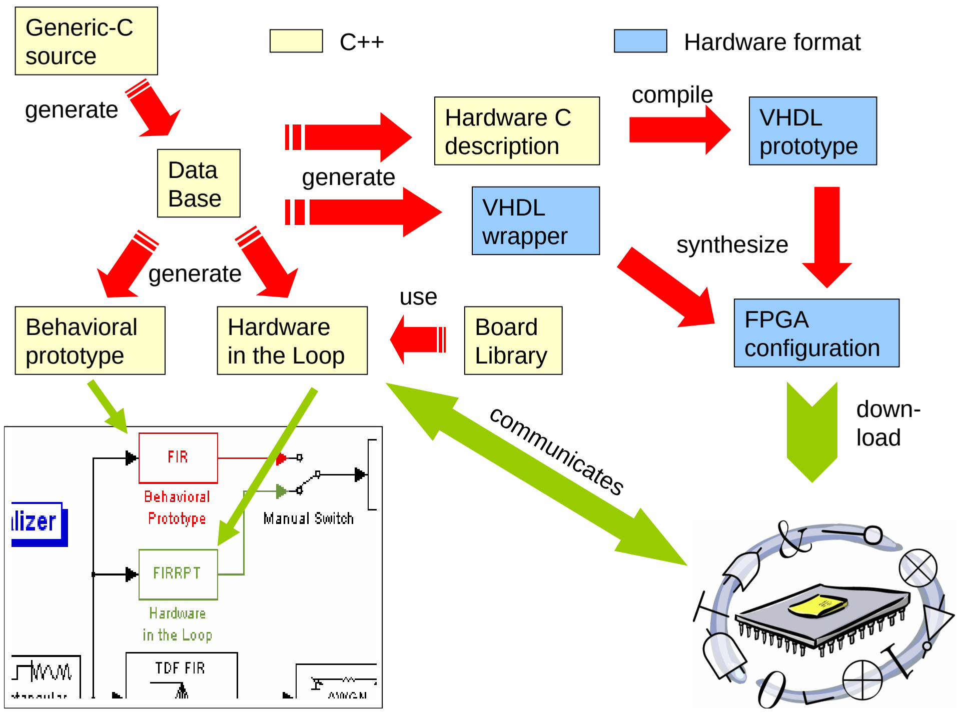


Figure 4.11: Refined design flow, including details.

Chapter 5

Implementation

By means of the refined design flow, presented at the end of Chapter 4 in Figure 4.11, prototyping tasks are identified and presented in a tool chain, the realized design flow. Its elements, these are the programmed tools and libraries, are described in detail. Finally the graphical user interface, integrating all tools in an application, is introduced.

5.1 Realized Design Flow

On the basis of the refined flow in Figure 4.11 a tool chain was established, described in this section. Each arrow, indicating data transformation, implies that a tool has to take on the corresponding task. Figure 5.1 illustrates the final, complete design flow, including all tools, both internal and third party. The flow begins in the upper left corner with the Generic-C source. It is analyzed by the parsing tool `GCParser`, saving its results in a database file, constituting the input for generation tools. Then the flow splits up in two branches, equal to the simple flow in Figure 4.1:

1. Simulink code generation and compilation.
2. Hardware generation, implementation and download.

In the first branch the database is read by `GCWriter`, which generates the prototype module. It is compiled, along with the behavioral and the "Hardware in the Loop" S-Function template, by Microsoft's `Visual Studio C++ compiler` into the respective Simulink blocks. Hardware is generated (in the second branch) by creating a hardware C source for A|RT Builderand converting it into the VHDL prototype. `VHDLWriter` reads both the database and the VHDL prototype in order to generate a VHDL wrapper. The VHDL top shell, along with the embedded wrapper and prototype, is synthesized by Xilinx' ISE and further implemented as FPGA bitstream. All implementation constraints and other required sources are taken from a template project.

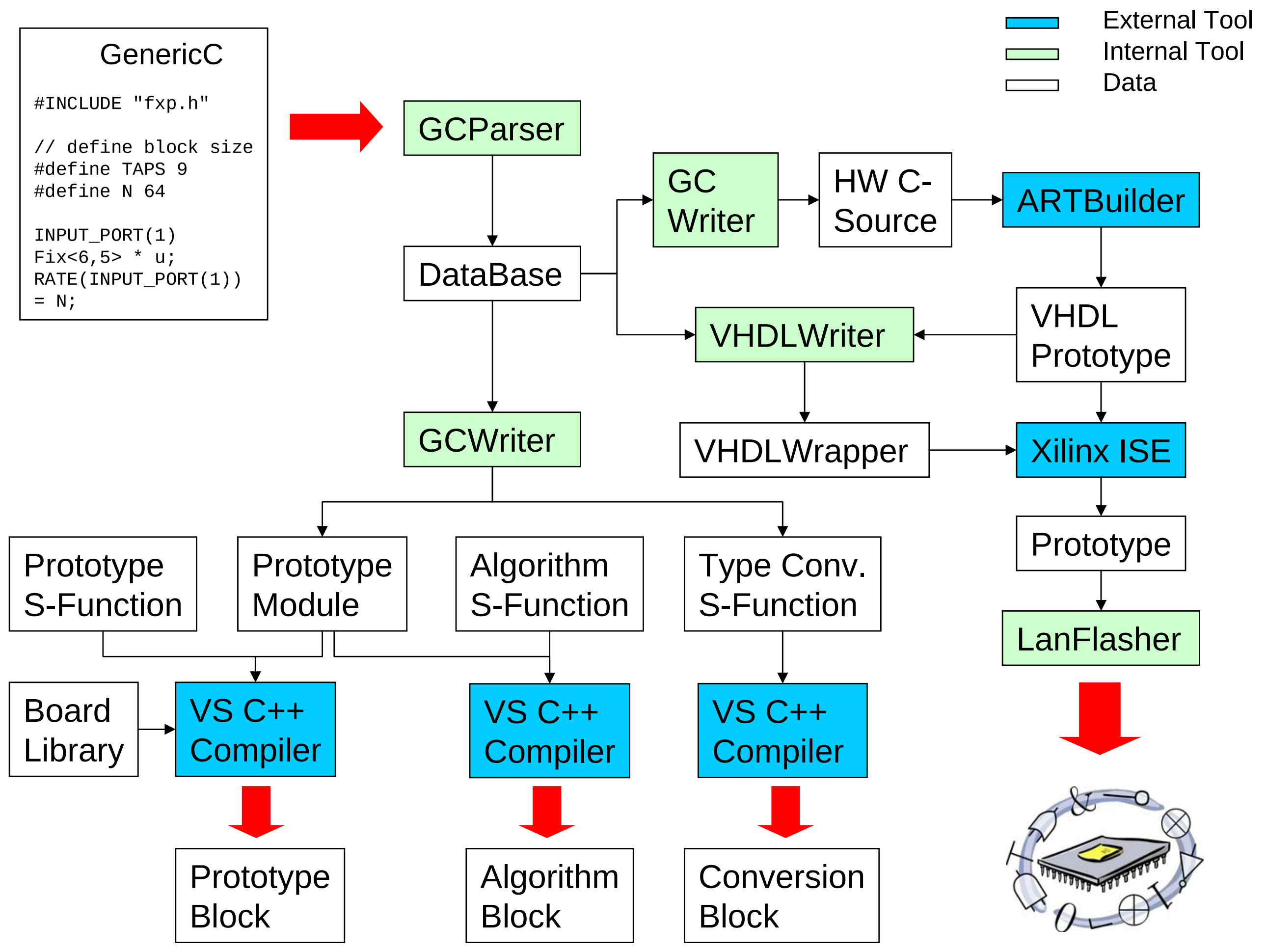


Figure 5.1: Rapid prototyping design flow.

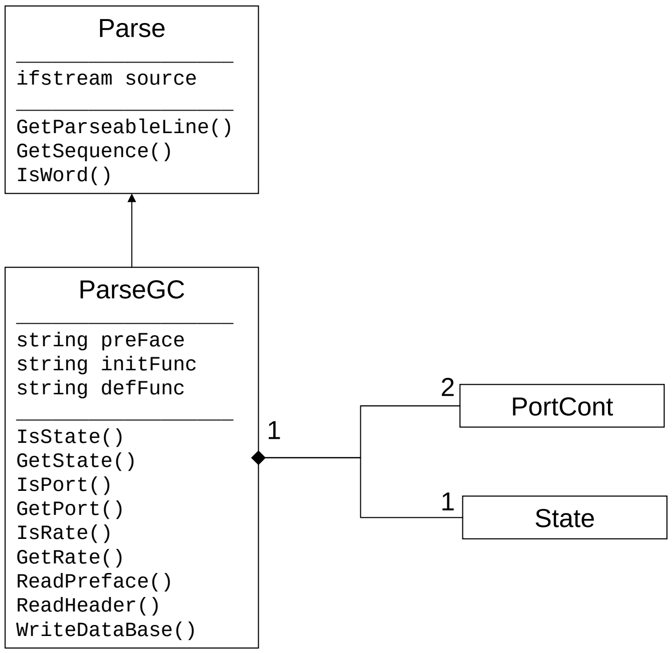


Figure 5.2: Class diagram for `GCParser`.

Finally the FPGA is reconfigured without a board reset by the ARCS tool `LanFlasher`, which loads the bitstream into the SDRAM (see Section 3.1.1) and activates the DSP to update the FPGA.

In the following all internal tools are described. Those implemented by myself feature the following common attributes:

- Written in object oriented C++.
- Tool wide C++ exception handling.
- Usage via command line.

5.1.1 Generic-C Scanner-Parser

Since the prototyping approach involves analyzing a programming language, a Generic-C parsing tool, called `GCParser`, is used. I decided to write it in object oriented C++ since I am experienced with it and useful stream processing libraries are provided. A first implementation in a single class turned out to be very faulty, since the amount of code got huge. In an evolution-like process I separated input based string processing from higher level token based syntax evaluation. The developed software model is illustrated in Figure 5.2.[1] As shown in the diagram, the actual Generic-C parser `ParseGC` inherits from a general parser `Parse`, providing scanner like string

[1]Please note that classes in the diagram contain only a selection of characteristic methods and members.

processing methods and basic parsing functionality, for instance analyzing numbers. `ParseGC` analyzes Generic-C statements and extracts their information. Generic-C IF information is saved in a state structure `State`, an input and an output port map `PortCont`. Finally the command line tool `GCParser` invokes `ParseGC`'s methods to analyze the Generic-C source and save its results in the database.

Scanning Part

First, `Parse` opens the input source file `source`, provided to the constructor. To keep memory usage low, the file is not read at once but in delimiter separated strings. Any character may be used as delimiter, however, my parser mostly uses carriage return line feed (CRLF), semicolon and braces. They are required to read preprocessor instructions, statements and C++ blocks, respectively. While an input stream is read character by character, it is transformed into a token string (not *stream*). I defined it to be a string with at least one space character between each token in order to allow for distinguishing them. Since in C operators, or anything else being not a word,[2] may not be separated by white spaces, token recognition for further processing is enabled by insertion of token delimiters (space characters). Whensoever an operator is recognized, a token delimiter is inserted. Since some operators are a subset of another operator ambiguities may occur, e.g. in this operation: `a+++b;`. `Parse` performs left to right scanning, i.e. the mentioned operation is evaluated as `a ++ + b`, however, evaluation as `a + ++ b` is syntactically correct, too. Each alphanumeric character and the underscore character are considered valid for identifiers and hence they are not separated. Furthermore, valid C++ line comments `//` and section comments `/* ...*/` are removed from the input stream such that all input to the parser consists solely of valid tokens. Since the parser operates on tokens its input should be a sequence of token-objects, rather than the string with tokens. Thus the intermediate token string format is transformed into a token stream, represented by a C++ vector of strings, which offers superior Standard Template Library (STL) functionality, like iterator access, for further processing by the actual (Generic-C) parser. Moreover special functionality for verbatim extraction of C function definitions is included. A function definition is all source code starting from the first left brace to the corresponding right brace ({ ...}). This verbatim extraction is required to generate the prototype module, the HW C source for A|RT Builder and to read code from template files.

[2] Any alphanumeric string (including underscore characters), whereas the first character is not a decimal digit (0 - 9).

Generic-C Parser

Expanding the general parser `Parse`, this one adds Generic-C specific syntax evaluation. Eleven Generic-C keywords are recognized by `ParseGC`, whereas *Prototype* is the Generic-C file name without extension:

1. `INCLUDE`
2. `DEFINE`
3. `INPUT_PORT`
4. `OUTPUT_PORT`
5. `RATE`
6. `STATE`
7. `RAM`
8. `DRAM`
9. `PARAMETER`
10. *`Prototype`*
11. `init_`*`Prototype`*

When such a keyword is found at the begin of a token stream, the corresponding Generic-C statement is assumed and evaluated. However, `INCLUDE` and `DEFINE` are Generic-C preprocessor keywords, why they must be preceded by `#`. For parsing, the Generic-C description has been split in two parts, illustrated in Figure 5.3:

1. Preface.
2. Generic-C description.

The "preface" contains all code ranging from the begin of the file to the first Generic-C statement (not Generic-C preprocessor instructions mentioned above). The separation in two parts is required in order to inherit comments from the preface in generated code, improving its readability. Thus, the preface is pasted verbatim into generated code, except that only Generic-C preprocessor keywords are translated to their lower case counterpart.

Once the preface has been read the parser expects Generic-C statements and function definitions only, starting with one of the above mentioned keywords, except `INCLUDE` and `DEFINE`. Section 4.2.1 provides a detailed lineup of fully supported Generic-C statements, described in EBNF. These statements form the specification for `ParseGC`. Compared to the keywords listed above no statements for `RAM`, `DRAM` and `PARAMETER` are provided, since they are not supported by RPT (for reasons refer to Section 4.2.1). When `GCParser` encounters one of these keywords it issues an error like: "currently not supported". Whenever a supported statement occurs, it is analyzed and only its information is stored in the parser's *database* objects. The database

```
#INCLUDE "mylib.h"
#INCLUDE "fix.h"

// define size
#define N 128

// constants for X
#define THIS 13
#define THAT THIS/3

// this is the last "preface" line
INPUT_PORT(1) Fix<16,15> * u;
OUTPUT_PORT(1) Fix<16,15> * y;
STATE int x [THIS];

RATE(INPUT_PORT(1)) = N;
RATE(OUTPUT_PORT(1)) = 2*N;

init_X()
{
    for (int i = 0; i<(THIS); ++i)
        x[i] = 0.0;
}

X()
```

Figure 5.3: Generic-C partition in header and preface.

```
// check for GenericC keywords
switch (IsKeyWord(*itor))
{
case INPUT_PORT:
    if (!IsPort(itor, INPUT_PORT))
        throw PARSEGC_ERR+4;
    port = GetPort(itor);
    // add successfully read port to data structure
    if (!(mInPorts.first.insert(port.index, port)))
        throw PARSEGC_ERR+5;
    if (!(mInPorts.second.insert(port.identifier.c_str(), port)))
        throw PARSEGC_ERR+6;
    break;
```

Figure 5.4: Generic-C parsing methodology.

includes all information from the Generic-C file in a uniform representation, presented in detail in the following Section 5.1.2. Figure 5.4 illustrates the parsing methodology by example code. For each supported Generic-C statement exists a method to test the statements syntax and a method to evaluate the statement, e.g., `IsPort()` and `GetPort()` for both `INPUT_PORT` and `OUTPUT_PORT`. These methods get a token stream as input and are implemented according to the statement's EBNF, whereas `IsPort()` returns true or false and `GetPort()` returns a `Port` object. However, `GetPort()` is invoked only once the statement's syntax has been verified, ensuring that only valid `Port` objects exist. Eventually the port is added to the database (`mInPorts` in Figure 5.4), which is immediately checked for consistency, i.e., the port *index* and the port *name* must be unique. `mInPorts` is of type `PortCont`, as depicted in Figure 5.2. As shown by the cardinality a second `PortCont` is used for output ports.

Similar to the `INPUT_PORT` statement, presented in Figure 5.4, all other statements are analyzed. The `State` object in Figure 5.2 is created, when the `STATE` statement is encountered. When it occurs a second time an error is issued, according to the Generic-C definition in Section 4.2.1. A `RATE` statement updates an existing port's data rate or causes an error when the port does not exist. Generic-C functions are added verbatim to the database. Once parsing has been accomplished the database is saved in a file in order to make it persistent.

User Interface

Thrown exceptions are caught in `GCParser`'s main function, where an according error message is issued on the standard error output. `GCParser` takes the Generic-C source filename, including the `.gc` extension, as input parameter and produces an equally named output file with `.dat` extension in the working directory.

5.1.2 Database

As shown in Figure 5.1 the database is created by `GCParser` and input by `GCWriter` and `VHDLWriter`. A Generic-C description grants various degrees of freedom and can contain errors, because it is human-created. Purpose of the database is to describe a Generic-C prototype in a uniform, flawless representation in order to allow generator tools being implemented straight forward. Additionally it is persistent (because it is a file) which allows for processing it at a later time. The database models the Generic-C IF, state variable, functions and the preface. Hence, it contains all information from the Generic-C source.

In order to be created or used the database must also exist in a runtime representation, or, in other words, as variable(s) in the tool's memory. To

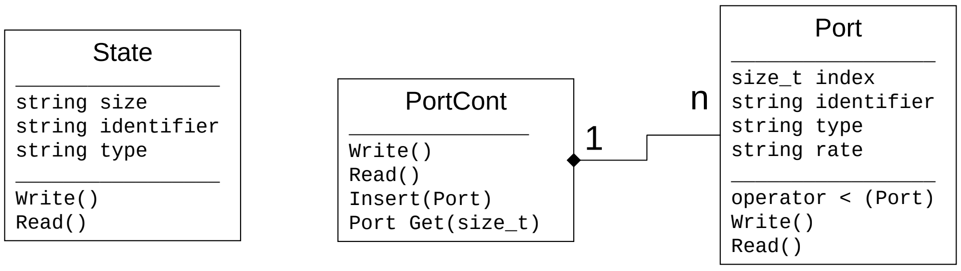


Figure 5.5: Database classes.

this end I use a `State` structure for the state, strings for the Generic-C functions and a `PortCont` container for input and output ports as depicted in Figure 5.5. `PortCont` is a unique pair associative container, i.e., all elements are associated with a unique key. I used an STL `map` as unique pair associative container, since it allows insertion and selection of elements by their key and provides mature, standardized, superb access methodology. Detailed information on the STL is provided by Silicon Graphics, Inc. in [28]. The element type of the `map` is `Port`, representing a single Generic-C port, consisting of an index, a data type, an identifier and a rate. `Map` is also a sorted container, thus requiring its element type to provide a strict weak ordering, i.e., `Port` provides the operator `<`, which compares two ports by their indices. The key, associated with a `Port`, is again its index. Thus, per definition of unique associative containers, there cannot be two ports with the same index in `PortCont`. Since port identifiers have also key character, a second `map` ensures that port identifiers are unique. Thus, actually a `PortCont` is a `pair` of two `map`'s, allowing to select a port by either its name or its index. As shown in Figure 5.2 two `PortCont` instances are used, since input ports and output ports are separated and have their own indices. In order to be persistent, as previously mentioned, a `PortCont` object can be written to a file by the `WritePortMap()` and reconstructed by `ReadPortMap()`.

`State` is a simple structure, comparable to `Port`. It contains the Generic-C states data type, identifier and array size. As can be seen on the basis of the `STATE` EBNF presented in Section 4.2.1 it must always be declared as array. Since at most one state variable per prototype can exist, no container is required.

A database is introduced in order to provide a uniform, consistent input to generator tools. It contains all information from the Generic-C source: input and output ports, initialization and definition function, state and preface.

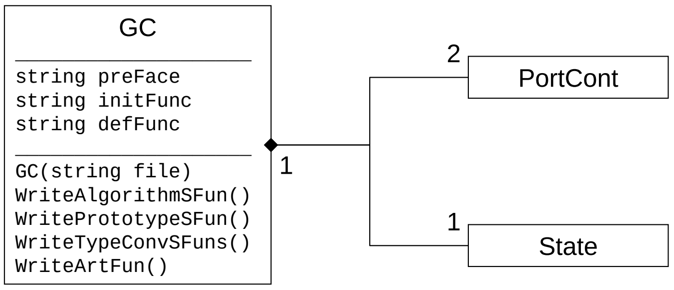


Figure 5.6: C++ generator class diagram.

5.1.3 C++ Generator

As illustrated in Figure 5.1 all Simulinkblocks and the HW C source for A|RT Builder are generated by the C++ generator `GCWriter`. It inputs the aforementioned database files. Actually `GCWriter` invokes an object of type `GC` to generate output, which is illustrated in Figure 5.6. Upon construction of a `GC` object the database file, specified in the constructor, is read and the runtime database representation is established. Then the four functions of `GC`, listed in the class diagram, can be invoked to generate output.

Behavioral Prototype S-Function

The function `WriteAlgorithmSFun()` generates all C++ code required to create a behavioral prototype block. According to Figure 5.1 this is the prototype module and the algorithm S-Function. The prototype module provides a prototype class with a static IF, i.e., it can always be accessed in the same way, independent from the Generic-C source. This results in the important fact, that the algorithm S-Function may be copied from a *template*. This template file, named `_GC_SFunTmpl.cpp`, and all other template files are located in a dedicated template folder mentioned in Appendix B. In general, the S-Function template is written generic, such that it can be compiled with any prototype module, regardless of the number and size of I/O and state. This is realized by means of preprocessor constructs, which use definitions (`#define`) in the prototype module to adjust the S-Function. As already described in Section 3.2.1, S-Functions use callback functions to integrate user functionality. The callback `mdlInitializeSizes()` defines the IF to the simulation model, including number, size and type of input and output ports. Figure 5.7 shows the template's code for defining the inputs of the block (outputs are defined similarly). The uppercase words, preceded by `_GC_`, are preprocessor constants. The array `inPortWidths` contains each input port's data rate and thus the template can realize any number of inputs with each having a different size or, in Generic-C termi-

```
    size_t i;
    if (!ssSetNumInputPorts(S, _GC_IN_PORTS)) return;
#if _GC_IN_PORTS
    size_t inPortWidths [_GC_IN_PORTS] = _GC_IN_PORT_RATES;
    for (i = 0; i<_GC_IN_PORTS; ++i)
    {
        ssSetInputPortWidth(S, i, inPortWidths[i]);
        ssSetInputPortDirectFeedThrough(S, i, 1);
    }
#endif
```

Figure 5.7: Template S-Function simulation interface code.

nology, a different data rate. Note, that even zero inputs are possible, which enables the realization of pure source blocks, e.g., waveform generators. A similar methodology as exemplified in Figure 5.7 is applied in the callback for the block outputs: `mdlOutputs()`. Simulink input data, output data and state pointers are handed over to the prototype, which creates a new set of outputs:

```
    Prototype p;
    // create a new set of outputs in ssOutPorts
    p.Do(xdisc, ssInPorts, ssOutPorts);
```

The prototype class and its methods are explained in detail in the following section.

Summarized, the behavioral prototype S-Function is compiled from a template S-Function and a generated prototype module, containing the prototype's functionality in form of a class.

Prototype Class

The prototype class, named `Prototype`, represents the core of the behavioral prototype. It contains the function definitions from Generic-C and provides a unified IF to the behavioral S-Function.

The skeleton of this prototype class is independent from Generic-C. Although it could be integrated into the program code in form of string constants I decided to put this skeleton into a template module, alike the S-Function templates. This allows to modify prototype generation without recompiling `GCWriter` just by editing the template module. This template module consists of two template files: a header file and a definition file. The header template, named `_GC_PrototypeTmpl.h`, contains a C++ class declaration statement, offering two public methods: `Do()` and `Init()`. These are the ones used in the S-Function callbacks mentioned above. `_GC_PrototypeTmpl.cpp` is the implementation template and contains the definitions of `Prototype`. The private methods `_Do()` and `_Init()` contain the original Generic-C code obtained from the corresponding functions. Recall from the Generic-C definition, that `_Do()` has implicit access to ports and state, though not listed as parameters. Thus the IF variables for `_Do()` are created from pointer arrays in `Do()`.

Basically two possibilities exist for prototype port and state memory management. Either Simulink memory may be used directly in the prototype or dedicated prototype memory is created. In order to enable both variants I selected pointers to implement the prototype IF. However Simulink memory may only be used directly, when data ports connected to the prototype block carry Generic-C data types. Since I use double precision floating point routing, data must be converted to the Generic-C type in the prototype class, which subsequently means that port and state memory has to be created inside the prototype class. The S-Function passes Simulink memory to `Do()` in form of arrays, whose size will be determined by generated constants. Once all port and state variables have been created, `_Do()` is called iteratively until it sets the completion flag `done`, as defined in the coding semantics in Section 4.4.2.

```
void Prototype::Do (double * _s,                              // state
                    double const * const * _ssInPorts [],     // input
                    double * _ssOutPorts [])                  // output
{
    // create Generic-C typed variables for _Do()
    ...
    while (!done)
        _Do();
    // write results into Simulink memory ...
    ...
```

The initialization function `Init()` analogously creates the state variable before calling the original Generic-C code in `_Init()`.

Hardware in the Loop S-Function

Relating to Figure 5.6 `WritePrototypeSFun()` creates all code required to compile the HIL S-Function. This, unlike in the behavioral S-Function, does *not* include any verbatim Generic-C function code.

It just copies the template S-Function `_GC_SFunRPTTmpl.cpp` and adapts names to the current prototype. This is again enabled by the generic style, already described above. The IF to Simulink is equal to the behavioral S-Function, introduced in Figure 5.7. However block behavior, more precisely the `mdlOutput()` callback, is different, since this blocks task is solely accessing the FPGA prototype. In order to access the HW platform and subsequently the FPGA via Ethernet I developed the `RPTPlatform` class, explained in Section 5.2.3 in detail. Amongst others it provides `RX()`, `TX()` for transmission and `GetArray()`, `PutArray()` for assembling transmission packets. Data transmitted by the library must already exist in binary format, i.e., one scalar port value is mapped into a 32 bit array. ANSI C types already are binary, however A|RT Library types are converted by the A|RT method `to_int()`. The function `SimulationToBinary()` assembles an array containing all binary input data for the FPGA and is called in the S-Function prior to data transmission. Since each input port has different types, `SimulationToBinary()` must be *generated*. Hence it is not placed in the template but in the prototype module mentioned above. When processed (binary) data is received from the FPGA the inverse functionality is required. `BinaryToSimulation` creates output port data from received data. In order to support all A|RT types I wrote the template function

```
template<typename T> T
PrototypeRPT::IntToFix (int val)
```

which creates any A|RT type `T` from a 32 bit value.

Fixed-Point Conversion S-Function

Sometimes it is desired to perform fixed-point conversion in an own block in order to observe quantization and overflow effects in the simulation model. Additionally, when A|RT Library types are used in model routings for prototype I/O, conversion blocks are also required. However, in the current version of `GCWriter` only `double` routing is used, which is Simulink default. Conversion functions are created from the `_GC_SFunFromTypeTmpl.cpp` and the `_GC_SFunToTypeTmpl.cpp` template. `WriteTypeConvSFuns()` is called to write a fixed-point conversion S-Function for each A|RT Library type found in the database. The S-Function name is created according to the name of the associated A|RT Library type, e.g., for a `Fix<4,3>` a file `DoubleFix_4_3_.cpp` is created.

A|RT Builder Source

As illustrated in Figure 5.1 `GCWriter` also generates the `HW C source`, by calling `WriteARTFun()`. This A|RT Builder source contains the behavioral function from the database. Since A|RT Builder creates the IF, i.e., the VHDL entity, from the functions parameter list, database ports are mapped there. Additionally `done`, introduced in Section 4.4.2, is appended to the list in order to allow routing the IR flag to the FPGA pins. Thus all variables, used in the behavioral function, have been declared, except the Generic-C `STATE` variable. Since it is initialized by a dedicated function, only declaring it is not sufficient. This initialization function must be called immediately when the variable is created, and, beyond it, the initialization must

```
INPUT_PORT(1) Fix<16,15> * u;
RATE(INPUT_PORT(1)) = N;

OUTPUT_PORT(1) Fix<16,15> * y;
RATE(OUTPUT_PORT(1)) = 2*N;

STATE Fix<16,15> x [TAPS];

init_SRRCOpt()
{
    for (int i = 0; i<TAPS; ++i)
        x[i] = 0.0;
}
```

```
struct _gc_TState
{
 Fix<16,15> x [TAPS];

_gc_TState ()
{
    for (int i = 0; i<TAPS; ++i)
        x[i] = 0.0;
}
};

_gc_TState _gc_State;
#define x _gc_State.x

void Prototype (bool & done,
const Fix<16,15> u [N],
Fix<16,15> y [2*N])
{
    #pragma OUT done
    done = false;
```

Figure 5.8: Generic-C IF and corresponding A|RT Builder source.

be mapped into the reset section of the sequential VHDL process. Fortunately A|RT Builder supports structures and their constructors by creating appropriate sequential logic. Hence, the state variable is mapped into a state structure and the initialization function into its default constructor. Figure 5.8 demonstrates the mapping concept with a small example.

User Interface

`GCWriter` reads the database file, specified by the corresponding input parameter *`Project`*`.dat`, and creates its output files in a subfolder `_gc_`*`Project`* of the current working directory. Outputs are:

- Prototype module: `_GC_`*`Project`*`.h`, `_GC_`*`Project`*`.cpp`.
- Behavioral prototype S-Function: *`Project`*`.cpp`.
- Hardware in the Loop S-Function: *`Project`*`RPT.cpp`.
- Hardware C source: *`Project`*`.cxx`.

Additionally the S-Functions for explicit type conversion are provided in the subfolder `_gc_TypeConversion` for each A|RT Library fixed point type used in the Generic-C IF.

5.1.4 VHDL Generator

VHDL Wrappers are generated in order to fit all possible prototypes with arbitrary port counts, sizes and types. As illustrated in Figure 5.1 this is

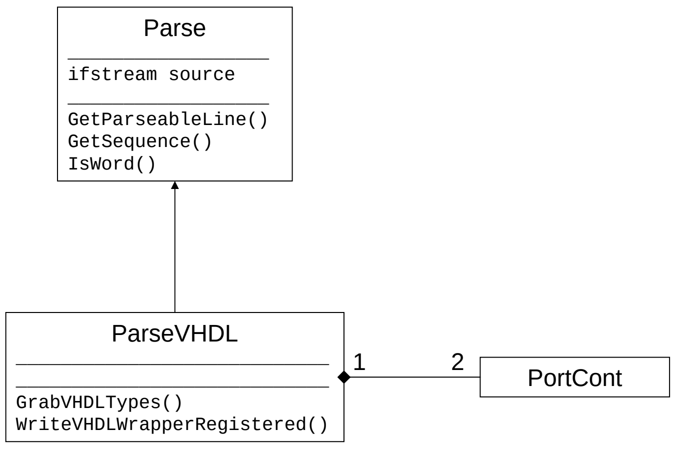


Figure 5.9: VHDL parser/generator class diagram.

performed by `VHDLWriter`, whereas it inputs both the database and the VHDL prototype. One may suppose that the database already contains all information to build the wrapper, however, VHDL types are unknown. It is more effort to build a tool generating A|RT Builder VHDL types, than scanning them from the prototype. Fortunately A|RT Builder inserts a port cross-reference list into the prototype comments, which associates each Generic-C type with the corresponding VHDL type.

Since analysis of this comments requires a lot of parsing functionality, already delivered by the general parser, I decided to reuse it as depicted in Figure 5.9. Since wrapper generation requires only input and output ports, the runtime database of `ParseVHDL` only includes the two `PortConts`.

First `VHDLWriter` creates a `ParseVHDL` object and invokes the method `GrabVHDLTypes()`. This method parses the cross-reference list in the VHDL prototype for VHDL type strings and adds them to the respective port container. By means of the expanded database the wrapper can eventually be generated by `WriteVHDLWrapperRegistered()`. The wrapper IF to the ARCS FPGA design is described in Figure 5.10. `iAddr`, `iData` and `oData` along with the control lines `iEnInput` and `iEnWr` constitute the IF to the CPU. Only a slice of the CPU address is provided to `RPTShell`, since bits beyond 19 are used in an external address decoder, which, among others, provides `iEnInput`. The last two signals are most important for controlling the prototype. `iInDataRdy` is a flip-flop set by the middleware. It enables the prototype and is reset as a side-effect of the prototype's "done" signal `oOutDataRdy`, which sets the output-ready IR flag. Since this entity is prototype independent only the architecture is generated.

Two sequential processes model the input and the output register files. The prototype is instantiated and connected to A|RT Builder typed input

```
entity RPTShell is
  port (
    iClk        : in std_logic; -- CPU clock
    iClkRPTCalc : in std_logic; -- prototype clock
    iRst        : in std_logic;

    iAddr : in  std_logic_vector(19 downto 0);
    iData : in  std_logic_vector(31 downto 0);
    oData : out std_logic_vector(31 downto 0);

    iEnInput    : in  std_logic; -- address decoding CS
    iEnWr       : in  std_logic; -- write enable
    iInDataRdy  : in  std_logic; -- data receive done
    oOutDataRdy : out std_logic  -- interrupt
    );
end RPTShell;
```

Figure 5.10: VHDL wrapper interface.

and output signals, which are mapped on the input and output register files, respectively. As mentioned in Section 4.4.3 the prototype has a distinct clock. Input port registers are written by the CPU, thus they are clocked by the CPU clock, unlike output registers, which are clocked by the prototype clock `iClkRPTCalc`. The latter is created in `top_config.vhd` and may be adjusted to individual prototyping needs. No synchronization logic is required between the two clock domains, since these clocks are created in a PLL which guarantees that they are synchronous. Additionally the prototype and the output registers are not enabled until all input data is registered. The input register file is 32 bits wide and its length equals the sum of all input port data rates. A|RT Builder's prototype inputs are mapped in the sequence, determined by the database file, on this array. This sequence is the same as used in the S-Function on the PC. Due to the input register size of 32 bits some bits are unmapped when prototype inputs are smaller. These unmapped bits of the input register file remain unused and are eventually discarded by the synthesis tool, thus no "dangling" flip-flops waste FPGA area. Similar to inputs, prototype outputs are registered and then mapped on a contiguous, 32 bit wide output memory map, wherefrom the board CPU reads words by indexing it with `iAddr`. Unfortunately, for big output register files, large MUX trees are inferred, which substantially contribute to the delay time and area consumption. This can be alleviated by using smaller output register files and subsequently smaller Generic-C data rates.

`VHDLWriter` takes two input parameters: a database *`Project`*`.dat` and a VHDL prototype. It generates the wrapper `RPTShell-RTL-a.vhd` in the subdirectory `_gc_`*`Project`*`_FPGA`.

5.1.5 FPGA Configuration

`LANFlasher` is a tool for the manipulation of the Flash memory attached to the DSP on the RTS-DSP board. It can read, write and erase the Flash contents using only the Ethernet interface of the RTS-DSP board (without requiring a JTAG interface) [17].

As presented in Section 3.1 this Flash is used to configure the FPGA during the boot process. Thus `LANFlasher` is used to save new FPGA configurations on the ARCS RTS-DSP board. However, attention must be paid to the FPGA configuration files. According to [17] `LANFlasher` requires a dual ported RAM (the `Common RAM`) in the FPGA for operation. If an FPGA configuration with faulty `Common RAM` communication is flashed, `LANFlasher` will not be able to access the Flash again. Fortunately `LANFlasher` has been extended with the ability to load FPGA configurations into the DSP's SDRAM, which in turn reconfigures the FPGA in seconds, i.e. without a board reset. Assumed an invalid FPGA configuration is used for the "dynamic reboot," a simple board reset restores the former FPGA status. RTS-DSP boards are identified by their rack and board identities (IDs), arising from the original application in channel simulator racks. `LANFlasher` requires these IDs and a source file (.bit extension) to reconfigure the FPGA. More details are provided in [17].

The design flow comprises the tools `GCParser`, `GCWriter`, `VHDLWriter` and ARCS `LANFlasher`. Furthermore it uses the third party tools A|RT Builder, Xinlinx ISE and Visual Studio C++.

5.2 Communication

When a `Hardware in the Loop` block is used in co-simulation it needs to access the FPGA. Figure 4.10 on page 42 illustrates this communication and involved layers. This section goes into more detail and presents the implementation of these layers bottom up for better understanding of design decisions.

Basically the entire communication concept is tailored to the FPGA prototype, since its overall aim is to transfer input port data to the prototype and get back output port data to the PC. As shown in Figure 3.1 data must be passed over the Coldfire micro controller (μC), in order to be transferred via Ethernet to the PC, where it is provided to Simulink with my prototyping library. Three identified communication sections are described in the following.

5.2.1 FPGA Prototype

Communicating with the FPGA prototype is comprised of two tasks. First, input port data is received from the Coldfire μC and, secondly, output port data is sent back to it. In order to handle arbitrary port counts a general scheme must be found. I decided to map all input ports on a contiguous input port memory map and equally all output ports on an output port memory map. Contiguous means, that ports follow each other directly in the addressing space, without unused space in between them. Hence all data can be downloaded to the FPGA in a single block bus operation. For data input the wrapper incorporates an address decoder, which enables just the correct register at write accesses. However, assumed a wrapper with RAM memories a contiguous memory map leads to HW overhead, since a complex enabling logic is be required, which can be avoided by address-aligned port memories. The μC's address space is partitioned as follows:

Registers FF000000h – FF0FFFFF: This space includes registers containing status information or parameters of importance to the middleware, such as "Input Request Size" or "Common RAM Size". These registers are all contained in the file `cold_reg.vhd`, located in the FPGA template project folder.[3] For rapid prototyping there are two registers of utmost importance: "FPGA Control" and "Coldfire IR." The former is located at FF000038h and provides bit zero as prototype enable to the wrapper. This flip-flop is connected directly to the wrapper input `iInDataRdy` mentioned in Figure 5.10. Once the middleware transferred all input data it sets `iInDataRdy` causing to begin prototype computation. Once the prototype finished it sets `oOutDataRdy`, which subsequently triggers the Coldfire external interrupt, located in "Coldfire IR" (address FF00001Ch) at bit zero. In parallel the prototype enable flip-flop is reset. The interrupt flag is, as customary, reset by the μC.

Further used registers for rapid prototyping are read-only and provide design parameters to the middleware as described in the following:

Input RAM Size: This read only register (at FF000028h) provides the size of the input memory and is used to allocate buffers in the middleware. It is set to the size of the wrapper input port memory map.

Output RAM Size: Analogously to the input RAM size the output RAM size (at FF00002Ch) contains the output memory size.

Input Request Size: (at FF000030h) contains the actual size of requested input data. Unlike the input RAM size this register is

[3]More information on tool structure is provided in Appendix B.

evaluated at runtime. However it is mirrored to input RAM size for rapid prototyping.

Output Available Size: (at FF000034h) is the output equivalent to the input request size.

Common RAM FF100000h – FF1FFFFF: The Common RAM, mentioned in Chapter 3, is not used directly for rapid prototyping. However, `LANFlasher` requires it for operation.

Input Memory FF200000h – FF2FFFFF: Input memories are written to this address space by the μC, up to the number of bytes contained in the Output RAM Size register. As previously mentioned they are concatenated to one contiguous space starting at address zero. This address space is *not* readable by the μC.

Output Memory FF300000h – FFF3FFFFF: This address space contains a prototype's output port memory map and is read-only for the μC.

According to the address space at most 2^{20} Bytes = 1 MByte of input and output memory, respectively, can be addressed. However, FPGA resources will set limits much closer anyway.

5.2.2 Middleware

The term "Middleware," widely used in this document, refers to the OS on the Motorola Coldfire μC. As shown in Figure 4.10 it interfaces the Ethernet and the prototype wrapper.

This OS is μClinux, customized for the Motorola Coldfire and described in [15]. It runs processes, which accept incoming TCP/UDP[4] Ethernet connections. These processes have been extended by to handle rapid prototyping data packets, distinguished by a particular identifier. When such a packet is found it is unpacked and its content is written into the FPGA's input memory section at FF200000h. A second packet identifier allows to write the FPGA control register to start the prototype. When the FPGA triggers an interrupt (IR) the output memory section is read and the IR flag is reset by the interrupt service routine (ISR). The Output Available Size register determines the number of bytes, which are packed and sent back to the PC.

5.2.3 Prototype Platform Interface

Accessing the FPGA is a vital for HIL simulations. I encapsulated the RTS-DSP board in the prototype platform IF providing functionality to access and control prototypes on a high level from S-Functions.

[4]transmission control protocol/user datagram protocol.

The class `RPTPlatform` provides all required functionality with a few member functions. Internally `RPTPlatform` uses the ARCS C library `NetPC` for communication tasks, more precisely, `RPTPlatform` is an additional communication layer comprised of the following methods:

`RPTPlatform (boardID, rackID):` This overridden constructor takes the rapid prototyping platform's rack ID and board ID. Since this constructor is the only one, ID's are always known after construction. It just creates the object in memory and does not call any functionality.

`Connect ():` This method is called first after creating an `RPTPlatform`. Since rack ID and board ID are known by construction no parameters are required to connect to the board. First `Connect ()` tries to discover the specified board on the net. When the board exists, a connection is established, and `RPTPlatform` data transfer methods may be called.

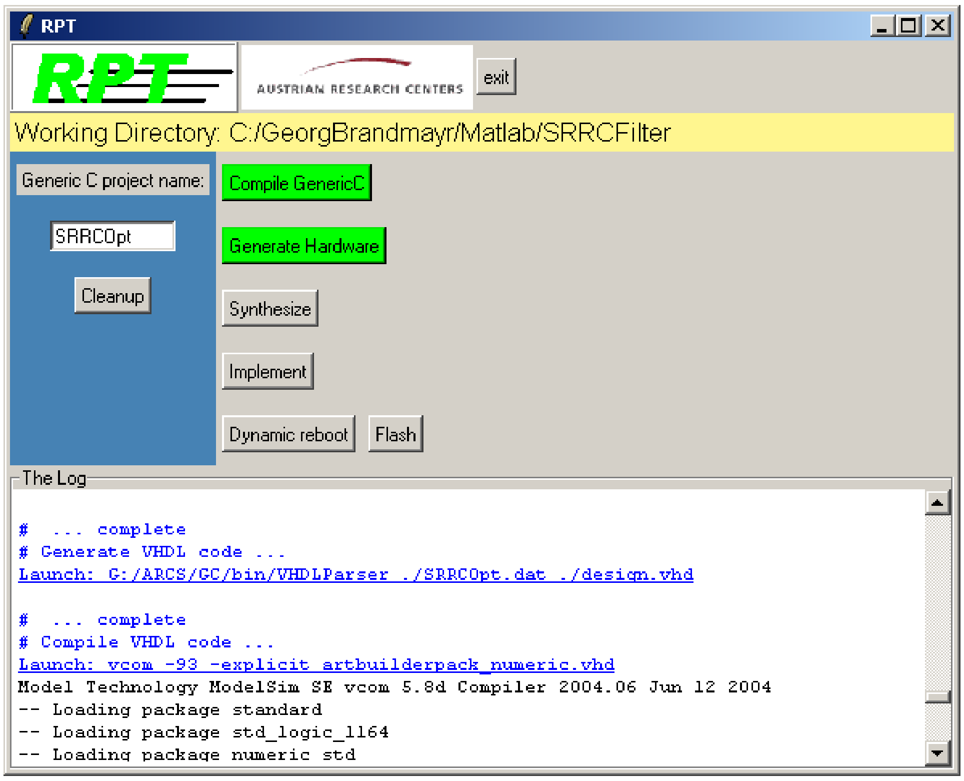
`GetArray (data, bytes, offset):` Prototype input port data is packed by `GetArray()` into `RPTPlatform`'s internal FPGA input data structure. It must contain a clone of the input port memory map as explained in the last sections. `GetArray()` was designed to copy a single input port at a particular position, so the parameter `offset` is required to indicate the position in the memory map. The data itself is determined by the pointer `data` and its size by `bytes`. An prototype with n ports requires `GetArray()` to be called n times.

`RX ():` Receive (viewed from the board) transfers the internal memory map to the board and then issues the prototype enable command, introduced above.

`TX ():` Transmit gets back outputs, which are placed in the FPGA output data structure. Actually no data is transferred via Ethernet during this call, since the middleware sends back output data immediately when an interrupt occurs. Thus `RPTPlatform`'s `TX()` just sends data from a buffer to the application and empties this buffer.

`PutArray (data, bytes, offset):` As a counterpart to `GetArray()` the method `PutArray()` copies an output port from the internal FPGA output data structure to user memory. The output port's position and size is specified by `offset` and bytes, respectively, and the target memory by `data`.

`ErrorMsg ():` The methods described above return with a code in case of an error. Additionally they create a human readable error message, which can be retrieved by calling `ErrorMsg()`.



Working Directory: C:/GeorgBrandmayr/Matlab/SRRCFilter

```
#  ... complete
# Generate VHDL code ...
Launch: G:/ARCS/GC/bin/VHDLParser ./SRRCOpt.dat ./design.vhd

#  ... complete
# Compile VHDL code ...
Launch: vcom -93 -explicit artbuilderpack_numeric.vhd
Model Technology ModelSim SE vcom 5.8d Compiler 2004.06 Jun 12 2004
-- Loading package standard
-- Loading package std_logic_1164
-- Loading package numeric_std
```

Figure 5.11: Graphical User Interface

Although I developed this class for use in HW in the loop S-Functions it may be used in any other C++ program as well, demonstrated with the `demo` example project mentioned in Appendix B.

Communication with the FPGA prototype is established by means of a board library on the PC, allowing to access μClinux on the RTS-DSP board. μClinux in turn accesses the FPGA wrapper and subsequently the prototype.

5.3 Graphical User Interface

Comfortable operation of RPT is enabled by a GUI, providing buttons for applying the presented design flow on Generic-C sources.

Initially RPT was controlled by MATLAB scripts, however, it turned out soon that scripts do not provide the demanded comfort. Furthermore the tedious launch of MATLAB just for compiling a Generic-C file displeased users. To tackle these problems I created a simple, quick GUI, presented in Figure 5.11. It provides six buttons for executing the design flow, presented in Figure 5.1, on a Generic-C file. Its name must be entered, without extension, in the `Generic-C Project name` entry. When enter is pushed and the file exists all buttons become activated and serve the following purposes:

Compile Generic-C allows to create a behavioral prototype and a HW in the loop block. First, `GCParser` analyzes the source and saves results in the associated database file. `GCWriter` is invoked to read the database and generate the prototype module. Finally, the Visual Compiler creates the MEX DLLs for Simulink by compiling the template S-Functions with the included prototype module. Refer to the left branch of the design flow tree in Figure 5.1.

Generate Hardware uses `GCWriter` to generate the C++ input for A|RT Builder, which subsequently generates the VHDL entity architecture pair. This VHDL description along with the database is used by `VHDLWriter` to generate the corresponding VHDL wrapper. All generated VHDL sources are compiled by ModelSim's VHDL compiler `vcom`. Refer to the right branch of the design flow tree in Figure 5.1. This button finally produces a synthesizable VHDL design.

Synthesize invokes Xilinx' synthesis tool to synthesize the VHDL design and output a net list.

Implement The net list is translated, mapped, placed and routed by the corresponding Xilinx tools as mentioned in Section 3.3.1. Finally the programming file for FPGA configuration is created.

Dynamic reboot uses `LanFlasher` along with the configuration file to reconfigure the FPGA without a board reset.

Flash updates the flash memory with the configuration file, which requires a board reset to update the FPGA.

When any of the buttons is busy, it starts to flash yellow. Eventually it switches to green, when it accomplished its task, or to red when a failure has occurred.

Output is logged in the window at the GUI's bottom. Execution logs of tools (external processes) are printed in black, whereas the associated command line instruction is printed underlined in blue, as exemplified in Figure 5.11. GUI outputs are printed blue and preceded by the GUI token `#`. Warnings and errors in execution logs are highlighted yellow and red respectively to allow for easy navigation in large logs.

5.3.1 Tcl/Tk

Tcl means Tool Command Language and Tk Tool Kit; the GUI is implemented in Tcl/Tk. Tcl is an interpreted language, i.e. a program interprets an ASCII source file at runtime. This is advantageous because, compared to binary executables, no compiling and building is required, allowing for quick modification of programs. These programs consist of commands, separated by line breaks. Each command consists of a name and one or more arguments, separated by white space. Tcl provides a rich set of commands for string manipulation, list and file processing. Furthermore it allows for easy launch and observation of external processes.

Tk is a graphics library providing window utilities like buttons, text windows or user entries. Theses so-called widgets are used as any other Tcl command. Geometry manager commands allow to automatically place and update these widgets. A Tk program (or GUI) is event based, i.e.the program is interpreted once at start up and *event handlers* are set up for widgets. These event handlers may again be Tcl scripts performing various actions. E.g., the push of a button may clean all files in a directory and print "cleaned." [25] is an excellent introduction to Tcl/Tk.

Tool Integration

In order to implement the design flow, the tools, described in Section 5.1, have to be coordinated. Tcl provides the `exec` command to call them as external processes. However, this command takes as long as the process lives. Placed in an event handler it blocks the event loop until the process terminates, making the GUI unresponsive. For processes with a runtime of milliseconds this does not matter, but HW implementation steps block for over 10 minutes. Of course these processes can be executed in the background, causing `exec` to return immediately and subsequently keeping the GUI responsive. Unfortunately all log info will be lost in this case, which is unacceptable.

Event-Driven Pipe

An event-driven pipe solves all above mentioned problems. It allows to execute processes in the background, while new log info triggers events whose handler prints the information in the GUI's log window. Thereto I developed the new command `Run`, taking a command line call as argument. It executes this command as pipe in the background, which is treated as file handler in Tcl. Then it sets up a file event handler, called whenever the pipe becomes readable. This file event handler prints the pipe's output on the log window.

5.3.2 Remote Operation

A|RT Builder runs on a Linux server and is unavailable for new installations. Thus RPT needs to access this server in order to call A|RT Builder.

The Secure Shell (SSH) is used to launch A|RT Builder and the secure file transfer protocol (SFTP) to upload sources to the server and download results from it. Putty, a free SSH client, and Psftp, a free SFTP client, perform these tasks. Accessing the server requires to log into an account, protected by a password. This password entry prevents full automated tool operation. However, with a private/public key pair and an authentication agent automation can be maintained. A password entry is required only once to activate the key in the authentication agent, called `Pageant`. Then both Putty and Psftp use this agent to access the server.

The implemented rapid prototyping flow and its tools GCParser, GCWriter and VHDLWriter, were presented in detail. Simulink blocks use a prototyping library to access the middleware, which, in turn, accesses the FPGA. The prototyping flow is controlled via RPT's GUI buttons, whereas tool outputs are collected in the GUI's log window.

Chapter 6

Application and Results

The prototyping environment's potential is demonstrated by it's application on an example. A typical wireless field simulation model is used for co-simulation with a behavioral prototype and the corresponding HW in the loop, automatically generated by application of Rapid ProtoTyper (RPT) on the model's SRRC transmit filter.

Assume a designer builds a radio frequency (RF) satellite link as illustrated in Figure 6.1. At the present design stage this model works perfectly fine. To get closer towards a product, for some processing-intensive parts the HW feasibility should be verified. The designer decides to first implement the SRRC transmit filter, since this is typically implemented in HW. An HIL simulation with an FPGA prototype would be optimal. To save valuable simulation time the SRRC Filter is put into a separate model, used for the co-simulations. Filters are linear time invariant (LTI) systems, which are characterized by their impulse response. Hence a Dirac sequence, applied to the various filters, is used for prototype verification in co-simulation. The corresponding model is depicted in Figure 6.2. According to the simulation semantics, defined in Section 4.3, data is processed in blocks whose size is determined by the Generic-C data rate. This results in distinct sample times once the blocks are unbuffered, as indicated by the sample time colors in Figure 6.2. For this example I chose a data rate (block size) of eight, as indicated by the signal dimensions in the figure. Smaller data rates result in less wrapper memory, which in turn alleviates the FPGA implementation effort.

SRRC filters are used to limit the required bandwidth of transmitted symbols in wireless systems by shaping them with a finite, cosine approximated pulse form. Figure 6.3 illustrates the general finite impulse response (FIR) filter implementation (tapped delay line) with the order M, which is described by

$$y(n) \quad = \quad b_0 x(n) + b_1 x(n-1) + \cdots + b_M x(n-M) \qquad (6.1)$$

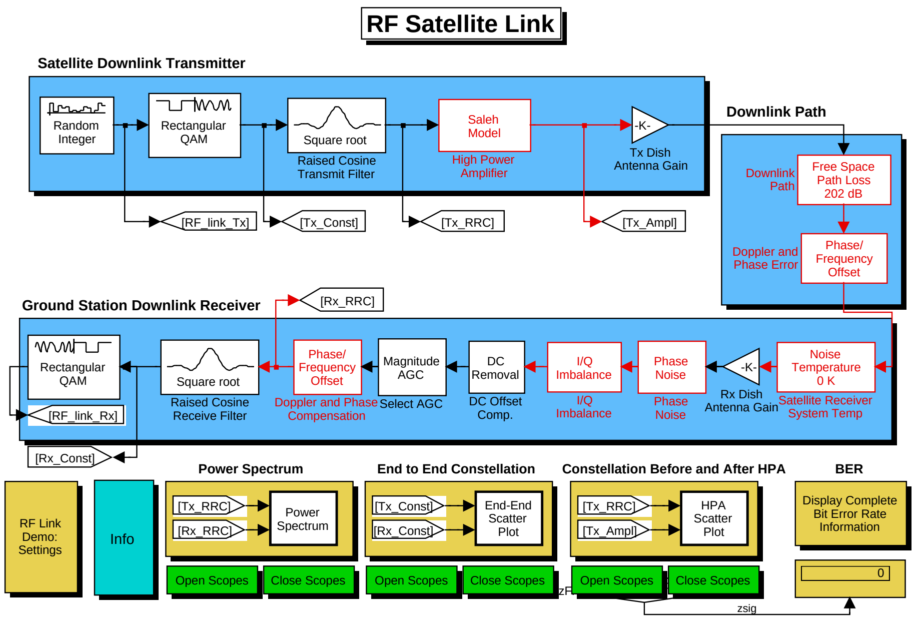


Figure 6.1: RF satellite link model (taken from the Simulink help).

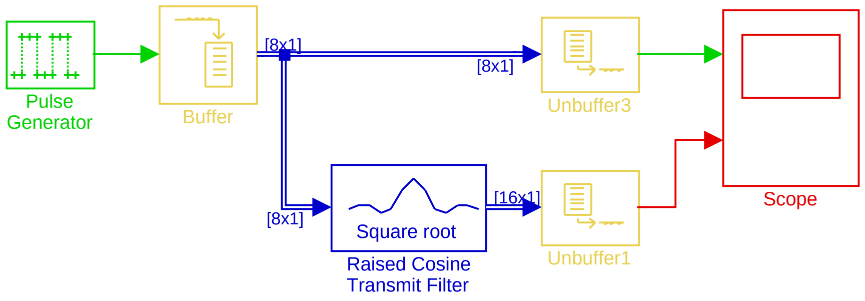


Figure 6.2: Simulation model for prototype verification.

$$= \sum_{m=0}^{M} b_m x(n-m) \tag{6.2}$$

According to (6.2) each output sample is a sum of $M + 1$ products (coefficients multiplied with the delay line's input samples). Generalized to a filter with arbitrary impulse response (any M and b_x) the output is a convolution of input and impulse response.

RPT is applied to a Generic-C implementation of the filter in order to create the Simulink blocks and the FPGA prototype. The complete

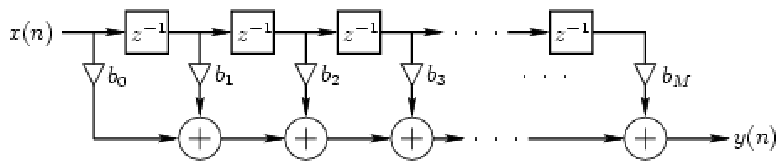


Figure 6.3: General finite-impulse-response (FIR) digital filter

Generic-C source and RPT's generated files are on the CD-ROM as listed in Section C.2. The following sections describe the typical usage of RPT in the provided sequence, viewed from the designers perspective.

6.1 Specification

First of all, before writing Generic-C code, a specification must be obtained for the targeted block, i.e., the SRRC filter. It is specified by the following parameters, taken from the transmit filter in Figure 6.1:

- Roll off factor: $\alpha = 0,18$
- Upsampling factor: $K = 2$
- Group delay (number of symbols): $T_g = 3$

The number of FIR coefficients $M + 1$ (length of the impulse response) is obtained with the filter's order:

$$M = 2 \cdot K \cdot T_g \tag{6.3}$$

Evaluation for the given specifications results in 13 coefficients. Simulink allows to save the filter block's coefficients to the workspace, wherefrom they can be obtained for Generic-C modeling.

6.2 Generic-C Source

A direct implementation of Figure 6.3 would lead to a filter requiring M unit delays and $M + 1$ multiply-accumulates (MACs) resulting according to the specification in 12 registers and 13 multiplications. Although this should fit easily into the FPGA (see Section 3.1.1), even if implemented fully parallel, the computational effort can be decreased by using mathematical optimizations, which is very probably closer to a product than the expensive direct implementation. A closer look at SRRC filters reveals that their impulse response is symmetric ($b_n = b_{M-n}$) and their number of coefficients $M + 1$ is always odd due to (6.3). Symmetry allows samples of identic coefficients to be firstly added and then the product to be built. Upsampling requires the insertion of zeros into the input signal. Since a product of zero and any coefficient will be zero, this implies that every K^{th} multiplication has to be performed. Additionally, since each K^{th} input value (including inserted zeros) is used, the length of the delay line reduces to $M/K + 1$.

Finally, a fixed-point representation of 16-bit width (fractional) has been chosen empirically. A second, decimated output port was added to the IF to have, at a time, an output with a data rate equal to the input. Three parts are used to present the source: the header, the initialization function and the algorithm function. The Generic-C header describes the filter interface:

```
#INCLUDE "fxp.h"  // A|RT lib

#define TAPS 13 #define DEL_LENGTH TAPS/2+1
// define block size
#define N 8

INPUT_PORT(1) Fix<16,15> * u; RATE(INPUT_PORT(1)) = N;

OUTPUT_PORT(1) Fix<16,15> * y; RATE(OUTPUT_PORT(1)) = 2*N;
// decimated output
OUTPUT_PORT(2) Fix<16,15> * y_dec; RATE(OUTPUT_PORT(2)) = N;

STATE Fix<16,15> x [DEL_LENGTH];
```

The delay line is implemented as the system's state and initialized by:

```
init_SRRCOpt() {
    for (int i = 0; i<(DEL_LENGTH); ++i)
        x[i] = 0.0;
}
```

The optimized filter's declarations:

```
SRRCOpt() {
    static Uint<8> index = 0;
    bool even = index[0] == 0;

    Fix<16,15> coeff_even [TAPS/4+1];
    Fix<16,15> coeff_odd [TAPS/4];
    coeff_even[0] = Fix<16,15>(-0.02584730312872);
    coeff_odd[0]  = Fix<16,15>(0.065168614718020);
    coeff_even[1] = Fix<16,15>(0.030577805550879);
    coeff_odd[1]  = Fix<16,15>(-0.13418905377870);
    coeff_even[2] = Fix<16,15>(-0.03369143649198);
    coeff_odd[2]  = Fix<16,15>(0.444705414395022);
    coeff_even[3] = Fix<16,15>(0.741884497481248);
```

Even $(b_0, b_2, b_4 \ldots)$ and odd $(b_1, b_3, b_5 \ldots)$ in variable names reference here to the coefficient indices, whereas duplicate coefficients have been omitted. The variable `index` is used to iterate over the port arrays; the flag `even` is used to switch coefficients.

```
    // delay line
    short i;
    if (even)
    {
        for (i = TAPS/2; i; --i)
            x[i] = x[i-1];
        x[i] = u[index>>1];
    }
    // filter output
    for (i = 0, y[index] = 0; i<TAPS/4; ++i)
    {
        Fix<16,15> coeff, symSum;

        if (even) // even
        {
           coeff = coeff_even[i];
           symSum = x[i]+x[TAPS/2-i];
        }
        else
        {
           coeff = coeff_odd[i];
           symSum = x[i]+x[TAPS/2-1-i];
        }
        y[index] += coeff*symSum;
    }
    if (even)
    {
        // unique coefficient
        y[index] += coeff_even[TAPS/4]*x[TAPS/4];
        y_dec[index>>1] = y[index];
    }
```

The delay line shifts only each second iteration since each second input value is a zero and, hence, contains no new information. Coefficient multiplexing (the `if` in the body) and symmetry utilization allow to reduce the number of required MAC units substantially.

```
    ++index;
    if (index == 2*N)
    {
        index = 0;
        done = true;
    }
}
```

This last algorithm fragment is required for termination. By setting the system flag `done`, programmers can determine when computations are completed. Since `index` is static its value is kept after the function `SRRCOpt()` returns, whereas it is invoked iteratively until `done` is set. Although all work

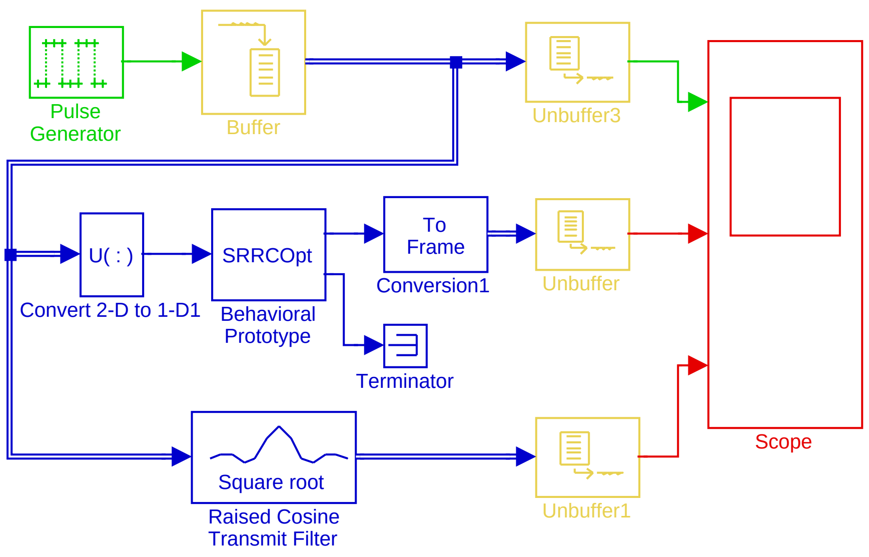


Figure 6.4: Simulation model with behavioral prototype in co-simulation.

can be done in one invocation, this would result in substantially larger HW prototypes, as explained in Section 4.4.2.

6.3 Behavioral Co-Simulation

The next step is to apply RPT on the completed Generic-C description in order to obtain a behavioral prototype for functional verification.

By pushing the "Compile Generic-C" button, as mentioned in Section 5.3, one obtains this behavioral prototype, which can be added to the previously presented verification model for performing co-simulation, as shown in Figure 6.4. The second, decimated is not used here to avoid overloading the model. Once the simulation is run, both impulse responses can be compared on the scope. They are equal except for the expected quantization error in the order of magnitude of 10^{-5}, caused by the 16 bit fixed-point representation. Thus the behavior has been verified and HW can be generated for HIL simulations. Please note, that the behavioral co-simulation performed in this section is optional. If one is confident, FPGA prototypes can be generated immediately.

6.4 Hardware Results

Clicking the "Generate Hardware" button generates VHDL sources, as mentioned in Section 5.1. A|RT Builder's VHDL sources are hardly human readable, however subsequent synthesis results may be more interesting. Since the algorithm was optimized to use not more than four multiplications per iteration resource utilization is expected to corroborate this assumptions. The design's (top shell, wrapper and prototype) synthesis report summary indicates this being correct:

```
Device utilization summary:
---------------------------

Selected Device : 2v2000bg575-5

 Number of Slices:                   1738  out of  10752    16%
 Number of Slice Flip Flops:          986  out of  21504     4%
 Number of 4 input LUTs:             3173  out of  21504    14%
 Number of bonded IOBs:               113  out of    408    27%
 Number of BRAMs:                      12  out of     56    21%
 Number of MULT18X18s:                  4  out of     56     7%
 Number of GCLKs:                       3  out of     16    18%
 Number of DCMs:                        2  out of      8    25%
```

Synthesizing the entire design without wrapper and prototype yields the following performance:

```
Device utilization summary:
---------------------------

Selected Device : 2v2000bg575-5

 Number of Slices:                    315  out of  10752     2%
 Number of Slice Flip Flops:          373  out of  21504     1%
 Number of 4 input LUTs:              519  out of  21504     2%
 Number of bonded IOBs:               113  out of    408    27%
 Number of BRAMs:                      12  out of     56    21%
 Number of GCLKs:                       3  out of     16    18%
 Number of DCMs:                        2  out of      8    25%
```

Although results cannot be compared directly, because Xilinx Synthesis Tool (XST) performs hierarchy dependant optimizations, they provide a rough estimation on the wrapper's and prototype's resource consumption.

The synthesis timing estimation is important for conclusions whether the prototype can be driven by the prototyping clock. This clock can be adjusted to individual real-time requirements and defaults to a tenth of the board's CPU clock, that is 66/10 MHz. A look at the timing summary of the full design including wrapper and prototype asserts that the prototype can be operated at this frequency (indeed at even higher frequencies).

```
Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 10.807ns (Maximum Frequency: 92.530MHz)
   Minimum input arrival time before clock: 5.228ns
   Maximum output required time after clock: 6.953ns
   Maximum combinational path delay: 4.088ns
```

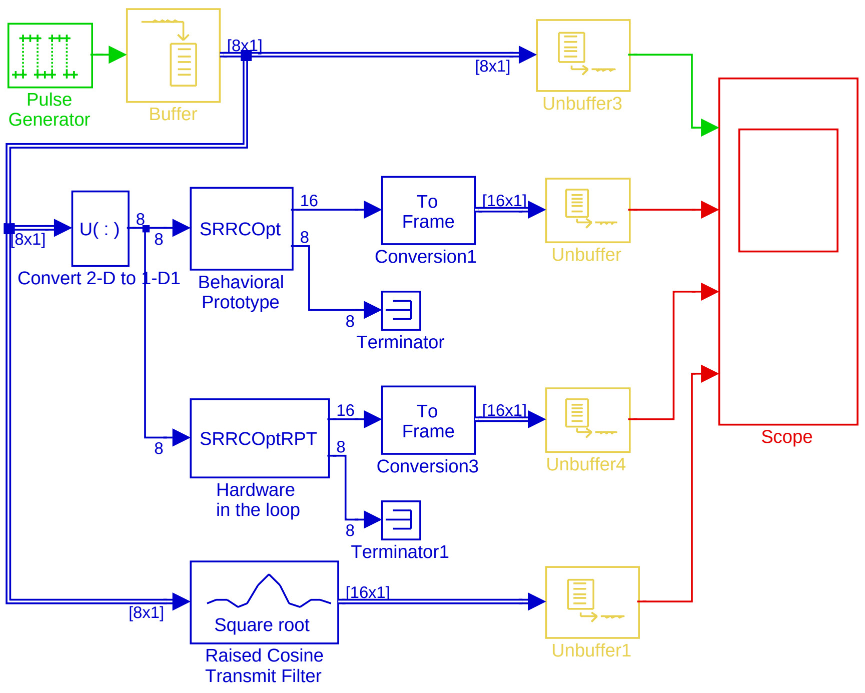


Figure 6.5: Behavioral prototype and Hardware in the loop co-simulation.

Although net delays are yet unknown and just roughly estimated by XST, the actual clock frequency, after place and route, should not be less than 6.6 MHz. Clicking the GUI's "Implement" button creates an FPGA bitstream and "Dynamic Reboot" reconfigures the FPGA on the RTS-DSP board with the new bitstream.

6.5 FPGA Prototype Co-Simulation

An HIL co-simulation verifies the generated FPGA prototype. Once more the verification model is expanded as shown in Figure 6.5. All filter representations are driven by the same source and have to produce the same result. Figure 6.6 approves this assumption, since all three impulse responses are equal, except for the mentioned quantization errors. The verified SRRC filter can be used to replace the original Simulink filter in Figure 6.1, and gradually further Simulink blocks can be replaced by their HIL counterparts.

RPT has been successfully applied on a real industry problem. An SRRC filter has been modeled in Generic-C and both the behavioral prototype and the FPGA prototype have been verified in co-simulation.

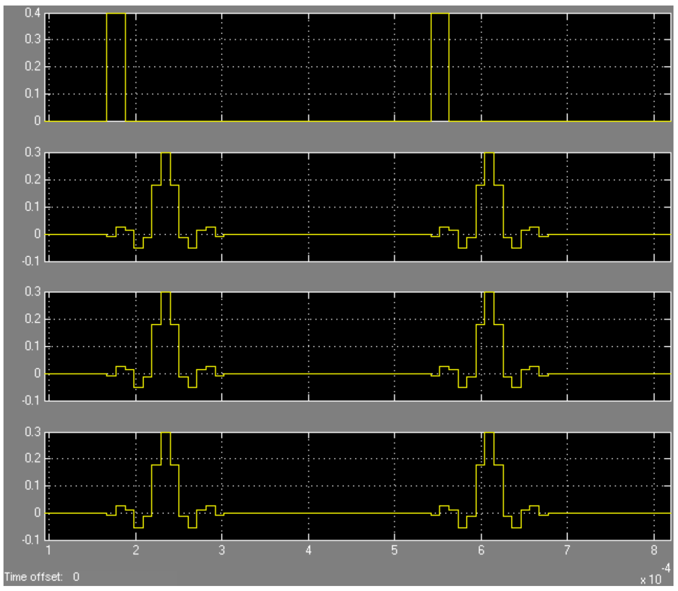


Figure 6.6: Upsampled filter impulse response for Figure 6.5.

Chapter 7

Conclusions and Recommendations

This chapter gives a summary on the work, showing its limitations and capabilities. In addition future optimizations, such as integration of better tools, are suggested.

7.1 Conclusions

A rapid prototyping design flow has been proposed, implemented in a tool, and proven by example. Other publications describe concepts of design flows which have similarities to the presented [27, 3, 5]. However, unlike these the presented design flow was implemented in a real, executable *tool*, emphasizing its realizability in a product, as claimed by ARCS. The prototyping flow requires mere the single source as input, except tool options, since all other sources for tools in the flow are *generated*. Compared to manual implementation, creation times for prototypes are substantially reduced. For instance, the application example presented in Chapter 6 was implemented in seven hours, including concept, coding and tool application.

The EDA industry has been struggling for years to increase design productivity. Typically, ASIC/FPGA designs are built by manual HDL coding. Assuming the amount of HDL code a designer produces is roughly constant over time, productivity is increased by decreasing the code size for a specific piece of HW. Behavioral synthesis, as presented in Section 2.3, is claimed to revolutionize HW design. However, this methodology has not been widely accepted in the EDA industry, since quality of results is insufficient. If designers think of using the presented prototyping tool for product design, the following should be considered.

C++ to VHDL translation is a crucial part in the design flow. A|RT Builder's VHDL code is huge in size and difficult to read, which makes it nearly impossible to understand or enhance it. Additionally, synthesis results are only suboptimal, since the abstraction prohibits the access of particular HW signals (e.g., flip-flop enable lines) which results in slightly higher resource consumption. Hence, this methodology is not applicable for highly optimized ASIC designs, produced in large quantities, since using fewer gates improves costs. Another concern are designs with high speed requirements and high resource utilization. It may not be possible to address all HW details in Generic-C, which is required to make the design fit into the given FPGA; or atomic C++ instructions may not be split up for register insertion to optimize critical paths. These problems still must be solved by providing manually optimized VHDL designs, which implies that the approach with A|RT Builder is not applicable for building optimized, ASIC like products.

Compared to behavioral synthesis tools like Catapult-C A|RT Builder is clearly at a disadvantage. Its “one to one” mapping concept requires an RTL like modeling style in C++ and thus nullifies any abstraction gains by using a high level language.

Nonetheless, designers, unfamiliar with VHDL but familiar with HW basics, such as latency, data dependency and parallelism, can use RPT to create FPGA prototypes. By using the GUI designers just have to create a Generic-C algorithm, from for instance a formal specification, and push buttons to invoke a 100% automated, comfortable tool chain. Although results are suboptimal, depending on HW design skills, a prototype can prove a design’s feasibility and exhibit some of a final product’s features as discussed in Chapter 1. Furthermore, the created HIL blocks allow for integration of FPGA prototypes into complex models, which in turn allows for comfortable FPGA design verification at a very high system level.

7.2 Recommendations

The above mentioned constraints are hints for future optimizations. Clearly, C-to-VHDL conversation has to be improved. Catapult-C from Mentor Graphics [21] seems to be the most promising tool to replace A|RT Builder. Another option, avoiding Catapult-C’s high license cost, is Impulse Accelerated Technologies’ CoDeveloper [11], which uses C-to-VHDL conversion technology from Stanford University.

Besides C-to-VHDL conversion, there is additional space for improvements. It would be useful to incorporate Simulink’s integrated fixed-point types as prototype I/O in S-Functions, since they are widely supported, for instance, Mentor’s “Link for ModelSim” uses these types.

In case of errors in HIL simulations, precious design time can be saved by performing co-simulation with a second behavioral prototype, comprising the prototype’s VHDL sources. If designers have cause for concern, they would start synthesis only once the co-simulation with the VHDL sources has succeeded.

As suggested in Section 4.4.3, there exist other concepts for wrapping the prototypes, differing in speed, area and flexibility. In particular the presented FIFO wrapper allows for substantially higher data rates, accelerating simulation. Since the presented flow offers only one wrapper, the others can be implemented, too, and selected by designers via tool options.

Finally, a very promising approach is to *unite structural* and *behavioral* synthesis in RPT. That means using the structure of a Simulink model, like Xilinx' System Generator [32] does, for interconnection of several Generic-C blocks and, subsequently, VHDL entities.

Appendix A

Installation

RPT was tested under Windows XP only and this installation procedure is therefore only applicable for this OS. The RPT executables are the intellectual property of ARCS and protected by copyright law. The reader might obtain a copy by contacting ARCS member Gerhard Humer (email: `gerhard.humer@arcs.ac.at`).

A.1 Requirements

RPT requires installations of:

- Xilinx ISE 6.3i or better.
- Microsoft Visual Studio 6.0 or better.
- Mentor Graphics ModelSim 5.5f or better.
- The MathWorks MATLAB 7.0.1 R14 SP1 or better.

A.2 Procedure

In order to install the prototyping system RPT all folders in the directory `/RPT`, as mentioned in Appendix B, have to be copied into a directory ***`RootRPT`*** on the PC. ***`RootRPT`*** must not contain spaces. The following modifications have to be carried out:

- Open the file ***`RootRPT`***`/RPT/tcl/RPT.tcl` and:
  - set the Tcl variable `matlabDir` to the MATLAB installation directory.
  - set rackID to your prototyping boards rack ID.
  - set boardID to your prototyping boards board ID.
- Open ***`RootRPT`***`/RPT/bin/mexopts.bat` and adjust the compiler settings:

  - Go to line 14 and set `MSVCDir` to ***RootVS***`/VC98` whereas ***RootVS*** is the Visual Studio installation directory.
  - Add ***RootRPT***`/RPT/inc` to the compiler include path variable `INCLUDE` on line 17.
  - Add ***RootRPT***`/RPT/lib` to the linker library path variable `LIB` on line 18.

- Open the properties menu of the link `RPT` in ***RootRPT***`/RPT/exmpl` and enter:

  ```
  RootRPT\RPT\bin\wish84.exe -f  RootRPT\RPT\tcl\RPT.tcl
  ```

  in the target entry. When typing in the string for *RootRPT* note, that windows accepts the \ directory separator only.

At the first start up of the tool the password for the SSH key must be entered (if the key is password protected). The authentication agent will keep the key file until it is closed. Without an activated key in the authentication agent no C-to-VHDL translation is possible. Nevertheless, behavioral prototypes can be generated server-independent.

A.3 Typical Usage

RPT itself and all its integrated sub-tools, as described in Figure 5.1, are designed to operate in arbitrary working directories (WD). Hence users typically choose their model's directory as WD since all S-Functions are required to be in this directory and our tool creates all final output data in its WD. Typically users place a link to the prototyping tool in the WD. This link starts the Tcl/Tk interpreter with my GUI script. As shown in Figure 5.11 the GUI displays its WD, which thus can be easily verified by users.

Appendix B

RPT

RPT's file structure and contents are described here. The structure is very similar to common tools as presented in the following.

Path: `/RPT/`

`bin/`	Binaries.
`doc/`	Documentation.
`exmpl/`	Generic-C examples for RPT application.
`inc/`	Includes folder with C++ sources.
`lib/`	Libraries for Tcl/Tk and C++.
`tcl/`	Tcl sources of the GUI.
`tmpl/`	Template folder with S-Function templates and an FPGA template project.

B.1 Binaries

Path: `/RPT/bin/`

`cat.exe`	Unix `cat` tool, used for input output piping.
`GCParser.exe`	The Generic-C parser and database generator.
`GCWriter.exe`	S-Function and C++ HW source generator.
`LANFlasher.exe`	Flash tool used for FPGA reconfiguration.
`mexopts.bat`	MEX options for S-Function compilation.

`pageant.exe`	Authentication client for SSH and SFTP protocols.
`humer_private_key.ppk`	Private authentication key file from Gerhard Humer.
`psftp.exe`	SFTP tool for uploads and downloads.
`putty.exe`	SSH client for remote operation.
`puttygen.exe`	Key generator/converter for SSH key files.
`VHDLParser.exe`	Prototype parser and VHDL wrapper generator tool.
`wish84.exe`	Tcl/Tk interpreter.

B.2 Documentation

Path: `/RPT/docs/`

`art_builder_mn_urf.pdf`	A|RT Builder user manual.
`art_library_mn_urf.pdf`	A|RT Library user manual.
`book.p1.pdf`	“Tcl and the Tk Toolkit,” part one.
`book.p2.pdf`	“Tcl and the Tk Toolkit,” part two.
`book.p3.pdf`	“Tcl and the Tk Toolkit,” part three.
`book.p4.pdf`	“Tcl and the Tk Toolkit,” part four.
`MBD05.pdf`	Model Based Design conference paper about RPT.
`Tkexampl.pdf`	Tcl example on an event-based exec logger.
`xst.pdf`	XST documentation .

B.3 Examples

Path: `/RPT/exmpl/`

`Faltung.gc`	Convolution of an arbitrary input with a 40 tap trapezoid ramp.
`Filter.gc`	Nine tap SRRC FIR filter.
`RectGen.gc`	Square wave signal generator, demonstrates source modeling.
`SRRC.gc`	SRRC transmit filter with symbol rate three, upsampling factor two and 13 coefficients. Consumes 12 multipliers. Outputs one sample per cycle.

`SRRCSimBehavioral.mdl`	Behavioral verification model. Co-simulates the behavioral prototype.
`SRRCSimHIL.mdl`	Extends the behavioral verification model with the HW in the loop block.
`SRRCOpt.gc`	Optimized SRRC transmit filter. Uses coefficient multiplexing and further optimizations. Consumes 4 Multipliers. Outputs one sample per cycle.
`TripleMUX.gc`	Multiplexes three inputs on a triple-rate output and on a decimated output. Demonstrates Generic-C multi-rate modeling.
`RPT`	Shortcut to the RPT GUI.

B.4 Includes

Path: `/RPT/inc/`

`fxp.h`	A|RT Library for fixed-point numerics include header.
`WinFPGARXTX.h`	ARCS RTS-DSP board access library header.
`WinFPGARXTX.cpp`	ARCS RTS-DSP board access library implementation.
`RPTLibDemo/`	A demonstration for using the RPT library in a C++ program.
`WinTools/`	The NetPC library from ARCS.
`Tools/`	Additional sources for the NetPC library.

B.5 Libraries

Path: `/RPT/lib/`

`fxpvcc.lib`	Visual Studio library of the A|RT Library.
`tcl8.4/`	Tcl library.
`tk8.4/`	Tk library.

B.6 Tcl Sources

The sources comprise a start-up file for the GUI, and a file for each button.

Path: `/RPT/tcl/`

`com.tcl`	VHDL compile script for ModelSim.
`comGC.tcl`	Tcl procedure for the GUI's "Compile Generic-C" button.
`confDyn.tcl`	Event handler for the "Dynamic Reboot" button.
`confStat.tcl`	Event handler for the "Flash" button.
`genHW.tcl`	Script file for the "Generate HW" button.
`imp.tcl`	Tcl procedure for the "Implement" button.
`RPT.tcl`	GUI start-up file. Creates all widgets such as the buttons and the log window.
`syn.tcl`	Tcl procedure for the "Implement" button.
`tkcon.tcl`	Jeffrey Hobbs Tcl/Tk console. Excellent for debugging or coding from scratch.
`wish84.exe`	Link to the wish executable in the binaries directory. Intended for use with the `tkcon.tcl` console start up file.

B.7 Template Files

Path: `/RPT/tmpl/`

`FPGATemplateProject/`	Contains all non-generated files for building an FPGA prototype. This project is not synthesizable on its own.
`_GC_PrototypeTmpl.cpp`	Prototype module implementation file template.
`_GC_PrototypeTmpl.h`	Prototype module header file template.
`_GC_SFunFromTypeTmpl.cpp`	Fixed-point type to double conversion S-Function template.
`_GC_SFunRPTTmpl.cpp`	
`_GC_SFunTmpl.cpp`	
`_GC_SFunToTypeTmpl.cpp`	Double to fixed-point type conversion S-Function template.

Appendix C

CD-ROM Contents

File System: Joliet[1]

Mode: Single-session (CD-ROM)[2]

C.1 Diploma Thesis

The diploma thesis PDFs and its LaTeX sources are contained in the root directory of the CD-ROM.

Path: `/`

`dt.pdf`	Diploma thesis for use with Acrobat Reader (PDF-File).
`dtPrint.pdf`	Diploma thesis prepared for printing (PDF-File).
`dt.tex`	Diploma thesis main LaTeX source.
`literature/`	Literature folder containing all bibliography items denoted with "Copy on CD-ROM" and additionally some of the remaining bibliography items.
`images/`	Contains all images of the thesis. Vector graphics exist in PDF format and pixel graphics in PNG format.
`images/sources/`	Graphics source files are placed in this folder, including PowerPoint files for block diagrams and Simulink models.

[1] *or* ISO9660 – for digital versatile discs (DVDs) accordingly different specs.
[2] *or* Multi-session (CD-ROM XA).

C.2 Example Files

This folder contains the wireless field SRRC example presented in Chapter 6.

Path: `/example/`

`SRRCOptRPT.dll`	Simulink HIL FPGA prototype access S-Function for co-simulation, generated by RPT.
`SRRCOpt.dll`	Simulink behavioral prototype S-Function for co-simulation, generated by RPT.
`FilterSim.mdl`	Simulink model file, used for co-simulation of the behavioral prototype and the HIL block.
`SRRCOpt.gc`	Generic-C source, containing the optimized SRRC filter.
`SRRCOpt.dat`	Database file, generated by RPT's GCParser.
`RPT`	Link for starting up RPT.
`SRRCCoefficients.mat`	MATLAB data file, containing SRRC filter coefficients generated by MATLAB's filter designer tool.
`GCCmd`	Temporary command file for batch operation of the remote tools (putty etc.).
`_gc_TypeConversion/`	Folder containing type conversion S-Functions.
`_gc_SRRCOpt_FPGA/`	Folder with the VHDL project.
`_gc_SRRCOpt/`	Folder with all generated C++ sources.

C.2.1 FPGA Project

Path: `/example/_gc_SRRCOpt_FPGA/`

`artbuilderpack_numeric.vhd`	A|RT Builder's fixed point implementation in HW.
`cold_reg.vhd`	Registers for the Motorola Coldfire CPU.
`top_shell.vhd`	Outermost VHDL shell, provided by ARCS.
`top_config.vhd`	Clock generation unit creates the prototyping clock among others,

	provided by ARCS.
`RPTShell-e.vhd`	Non-generated VHDL wrapper entity.
`RPTShell-RTL-a.vhd`	Wrapper architecture, generated by VHDLWriter.
`design.vhd`	Prototype, generated by A|RT Builder from the HW C source.
`top_shell.bit`	FPGA configuration file, ready for use with LANFlasher.

C.2.2 C++ Sources

Path: `/example/_gc_SRRCOpt/`

`SRRCOptRPT.cpp`	S-Function source code for the HIL block.
`SRRCOpt.cpp`	S-Function source code for the behavioral prototype block.
`_GC_SRRCOpt.h`	Prototype module header file.
`_GC_SRRCOpt.cpp`	Prototype module implementation file.
`SRRCOpt.cxx`	A|RT Builder's HW C source.
`mexopts.bat`	Compiler options for MATLAB's `mex` utility, which in turn passes them to the VC++ compiler.

Bibliography

[1] Adelante Technologies: *A|RT Builder User and Reference Manual*, v3.1 rev5 ed., 2002. Copy on CD-ROM: `art_builder_mn_urf.pdf`.

[2] Adelante Technologies: *A|RT Library User and Reference Manual*, v3.1 rev5 ed., 2002. Copy on CD-ROM: `art_library_mn_urf.pdf`.

[3] Adjoudani, A., E. Beck, A. Burg, G. M. Djuknic, T. Gvoth, D. Haessig, S. Manji, M. Milbrodt, M. Rupp, D. Samardzija, A. Siegel, T. Sizer II, C. Tran, S. Walker, S. A. Wilkus and P. Wolniansky: *Prototype Experience for MIMO BLAST over Third Generation Wireless System*. In *Special Issue JSAC on MIMO Systems*, vol. 21, pp. 440–451, 2003.

[4] Brandmayr, G., G. Humer and M. Rupp: *Automatic Co-Verification of FPGA Designs in Simulink*. In Ritt, M. (ed.): *Model-Based Design Conference*, pp. 21–30. The MathWorks, Shaker Verlag GmbH, 2005.

[5] Burg, A., E. Beck, M. Rupp, D. Perels, N. Felber and W. Fichtner: *FPGA implementation of a MIMO receiver front-end for UMTS*. In *Proc. International Zurich Seminar on Broadband Communications*, pp. 8_1–8_6, Feb. 2002.

[6] Celoxica: *Handel-C*. `http://www.celoxica.com/technology/c_design/handel-c.asp`, 2005. Copy on CD-ROM: `Celoxica05.pdf`.

[7] Easterbrook, S.: *How Theses Get Written: Some Cool Tips*. `www.cs.toronto.edu/~sme/presentations/thesiswriting.pdf`, 2003. Copy on CD-ROM: `thesiswriting.pdf`.

[8] EDACafe: *High-Performance Software Tool Recognized For Simplifying Hardware Verification Process*. `http://www10.edacafe.com/nbc/articles/view_article.php?section=CorpNews&articleid=119014`, 2004. Copy on CD-ROM: `EDACafe04.pdf`.

[9] GSMWorld: *One Million Customers Milestone*. `http://www.gsmworld.com/about/history/history_page12.shtml`, 2004. Copy on CD-ROM: `GSMWorld04.pdf`.

[10] Humer, G., H. Ritt and M. Meyenburg: *Antenne mit Intelligenz*. In *Embedded World Kongress 2003*, Nürnberg, Germany, Feb. 2003.

[11] ImpulseC: *From C to FPGA*. http://www.impulsec.com/C_to_fpga.htm, 2005. Copy on CD-ROM: `ImpulseC05.pdf`.

[12] Jaeger, J. and S. McCloud: *The four Rs of efficient system design*. EE Times, March 2005.

[13] Kaltenberger, F., G. Steinböck, R. Kloibhofer, R. Lieger and G. Humer: *A Multi-band Development Platform for Rapid Prototyping of MIMO Systems*. In *ITG Workshop on Smart Antennas*, Duisburg, Germany, April 2005.

[14] Kloibhofer, R. and H. Eberl: *Kanalsimulator V1.0 2003-11-10*. Techn. Rep., ARC Seibersdorf research GmbH., Vienna, Austria, 2003. Copy on CD-ROM: `Kanalsimulator_v1.0_2003-11-10.pdf`.

[15] Krzalic, S.: *μClinux auf dem Motorola Coldfire Evaluation Board*. Master's thesis, FH Technikum Wien, July 2004.

[16] Lall, N. and E. Cigan: *Plug and Play Design Methodologies for FPGA-based Signal Processing*. FPGA Journal, March 2005.

[17] Lieger, R.: *LANFlasher*, 1.0 ed., 2004. Copy on CD-ROM: `LANFlasher.pdf`.

[18] Maier, G.: *Prototypen Entwicklungsumgebung zur schnellen Simulation*. Master's thesis, University of Applied Sciences Hagenberg, Hagenberg, Austria, July 2004.

[19] McCloud, S.: *Algorithmic C Synthesis Optimizes ESL Design Flows*. Xcell Journal, Fall 2004.

[20] Meindl-Pfeiffer, G., R. Kloibhofer, F. Kaltenberger and G. Humer: *Multistandard Development platform for MIMO Software Defined Radio*. In *EUSIPCO*, Antalya, Turkey, Sept. 2005.

[21] Mentor Graphics: *Catapult-C Synthesis*. http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm, 2005. Copy on CD-ROM: `Mentor-Catapult05.pdf`.

[22] Mentor Graphics: *Leonardo Spectrum*. http://www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum/, 2005. Copy on CD-ROM: `Mentor-Leo05.pdf`.

[23] Meredith, M.: *A look inside behavioral synthesis*. EE Times, April 2004.

[24] Morris, K.: *Catapult C*. FPGA Journal, June 2005.

[25] Ousterhout, J. K.: *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, Inc., 1994.

[26] Rupp, M.: *GenC readme3*. Techn. Rep., TU Vienna, Vienna, Nov. 2001. Copy on CD-ROM: `GenC_readme.pdf`.

[27] Rupp, M., A. Burg and E. Beck: *Rapid prototyping for wireless designs: the five-ones approach*. In *Signal Processing*, vol. 83, issue 7, pp. 1427–1444, July 2003.

[28] Silicon Graphics, Inc.: *Standard Template Library Programmer's Guide*. `http://www.sgi.com/tech/stl/`, 2005.

[29] The MathWorks: *Writing S-Functions*. `http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/sfg/sfun.html`, 2005.

[30] Velasquez, C., M. Gauckler and M. Polasek: *Integrated FPGA Rapid Prototyping Environment*. Master's thesis, ETH Zürich, July 2004.

[31] Xilinx: *ISE 6 manual*. Xilinx, San José, USA, 2004. Copy on CD-ROM: `xst.pdf`.

[32] Xilinx: *System Generator Extended Feature Set*. `http://www.xilinx.com/products/software/sysgen/features.htm`, 2005. Copy on CD-ROM: `Xilinx05.pdf`.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —

Breite = 100 mm
Höhe = 50 mm

— Diese Seite nach dem Druck entfernen! —