AUTOMATIC CO-VERIFICATION OF FPGA DESIGNS IN SIMULINK

Georg Brandmayr[1], *Gerhard Humer*[2], *Markus Rupp*[3]

1: FH Hagenberg
Hauptstrasse 115
4232 Hagenberg, Austria

2: ARC Seibersdorf research
Donau-City-Strasse 1
1220 Wien, Austria

3: Vienna University of Technology
Institute for Communications and RF Engineering
Gusshausstr. 25/389, 1040 Wien, Austria

ABSTRACT

Verification of DSP systems is an error-prone and time-consuming process, because many manual steps are required to create a prototype. Rapid prototyping is an emerging key design methodology, which allows designers of DSP systems to quickly verify their algorithms by co-simulation. Our rapid prototyping system automates the creation of hardware prototypes defined by just a single input source in GenericC. To verify the feasibility of the design in hardware, prototypes are generated for a Xilinx Virtex-II FPGA, which is part of the RTS-DSP board from ARC Seibersdorf research. Matlab Simulink serves as platform for simulation and hardware co-simulation and thus accesses the prototype. An SRRC filter in co-simulation successfully demonstrates the potential of the rapid prototyping system. The time required for creation of a prototype is reduced by at least one order of magnitude compared to manually coded prototypes. Although area and speed rates of a prototype are typically worse than compared to hand-coded prototypes we show that it is possible to build a one-button rapid prototyping system.

1. INTRODUCTION

Trends in the electronic industry show that design complexity is increasing faster than ever before, although the time for development of a product is becoming smaller. Consider, for example, mobile phones in the wireless communications market. In the early 90's, when GSM was launched, the architecture of a mobile was straightforward [1]. A typical product consisted of numerous small integrated circuits, with roughly equal partition between RF and digital frontend. Besides phone calls and soon later SMS no substantial features were provided. But today's architectures of 3G devices show an above average complexity increase for the digital part, because the market demands for as much functionality as possible in a single device. System on a chip helps reducing the number of ICs and offers never seen complexity — complexity which designers have to deal with.

Prototypes can help unveiling design bugs, which might be hidden in the simulation stage because they allow to explore the behavior of a "real" product. In particular FPGA prototypes are suited to explore the behavior of hardware components. 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, like multipliers, block RAMs or DSP-slices. Even timing issues, 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 products 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. But a drawback of prototypes is that for nowadays high complexity systems their implementation is costly and time consuming. There is no use in a prototype when 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 completion of the product, which includes prototypes. Traditionally FPGA prototypes are implemented manually, which means that a hardware designer obtains a behavioral description of the prototype, e.g. a Matlab script, and tries to convert it into a HDL description. This is the same proceeding as for the product itself, which results in the above mentioned 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. This approach prohibits human errors from VHDL 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 product. Some of the aspects presented so far are not new. The five ones approach, introduced by Rupp in [2], recommends using one "golden" code as well as heavy tool automation. In [3] a rapid prototyping environment, very similar to ours, is depicted except that the one source paradigm is not used, since VHDL code is created manually.

As one may have figured out until now our approach provides another advantage: designers do not have to (immediately) deal with HDL any longer. But this is only true as long as there are no bugs in the generated hardware. In such cases it must be debugged, which still has to be done by hardware experts. But, compared to the situation some years ago, nowadays designers have valuable support in simulating hardware (more precisely HDL) by high level tools. E.g., Simulink has been recently extended with the award win-

This work has been funded by the Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms.

ning[1] capability to simulate HDL code directly in Simulink. Thus it is now possible to use all high level features of Simulink, like data representation on scopes, for handling and visualization of HDL simulation data. E.g., 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 FFT scope — features which are not provided by common HDL simulators. Again it is not necessary to deal directly with the HDL to perform the co-simulation. But 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.

The following sections introduce our automated rapid prototyping concept and the upcoming software. Section 2 gives an overview of the ARCS RTS-DSP board[2], which contains the FPGA yielding the prototype. Our rapid prototyping methodology, that means the design flow and its principles, is established in Section 3, along with the implementation, which includes the realized tools of the design flow and the GUI. Section 4 covers the successful application of our tool on a small, yet meaningful example: an SRRC filter.

2. ARC PLATFORM

The prototyping platform is the RTS-DSP board from ARC Seibersdorf research. It contains a DSP part for executing simulation algorithms and a CPU part which handles peripheral tasks like communication to external systems. Fig. 1 gives an overview of the board architecture. A detailed description of the board is provided in [5, 6].

2.1 DSP part

This part contains a TMS320C6416 DSP from TI, clocked at 600 MHz, which is suitable for complex computations and algorithms. The second processing element is a Xilinx VirtexII XC2V2000 FPGA. Its main purpose is to establish a connection between the DSP and the CPU part via dual port block RAMs. 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 boot record and the FPGA bit file. EMIFA connects 32 MBytes SDRAM to the DSP. If necessary the system can be upgraded with up to 256 MBytes of commercial SODIMM SDRAM [7].

2.2 FPGA

The dedicated dual ported RAM, mentioned in the previous section, handles the communication between CPU and DSP (Fig. 1). In general the remaining chip area is used for HW implementable, computational tasks which decrease the work load of the DSP. When the RTS-DSP board is used in conjunction with our tool 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 a.s.o.. 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 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 8 DCMs (digital clock manager). Interrupt pins of both DSP and CPU can be accessed by the FPGA, which allows for efficient data transmission to adjacent cores. 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.

2.3 CPU Part

CPU of the board is a Motorola Coldfire MCF5272 chip. Its main task is data transmission between FPGA and CPU and execution of the TCP/IP stack for the ethernet interface. It is clocked at 66 MHz and features an integrated FEC (fast Ethernet controller), which controls a fast Ethernet transceiver LXT971A from Intel. The transceiver carries out the ISO-OSI physical layer. The external memory interface (IF) of the CPU connects 16 MBytes of SDRAM, 4 MBytes of flash and the FPGA [7, 8].

2.4 Interfaces

The most important IFs for this project are:

Ethernet connector Enables data transmission to the host PC. This port, with its substantially lower data transfer rate than the LVDS IF, was initially intended to transfer only visualization data[3]. For this purpose its data rate is sufficient. 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. But small execution time of the simulation is not a primary goal of this project.

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

JTAG IFs enable direct programming of the FPGA and flash memories. A RS232 IF allows debugging of the CPU [7].

Once a new .bit file has to be loaded onto the FPGA it can be loaded via ethernet, CPU and FPGA, directly into the DSP SDRAM (see Fig. 1), wherefrom the DSP reconfigures the FPGA at runtime, i.e., no board reset is required.

3. RAPID PROTOTYPING METHODOLOGY

Imagine a rather complex simulation model, e.g., a MIMO BLAST system as implemented by [9], in a graphical, block based simulation environment like the widely known Simulink. At an early project stage the entire simulation will execute solely on the PC. To verify the feasibility of the design (for more reasons refer to Section 1) designers might

[1] The Mathworks and Mentor Graphics won the EDN Innovation of the Year Award for their co-simulation environment Link for ModelSim. This exclusive awards program honors outstanding electronic products, ranging from integrated circuits to test equipment, and the creative engineers who invent them [4].

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

[3] The ARCS RTS-DSP board was designed for a (mobile communications) channel simulator, for more information refer to [7].

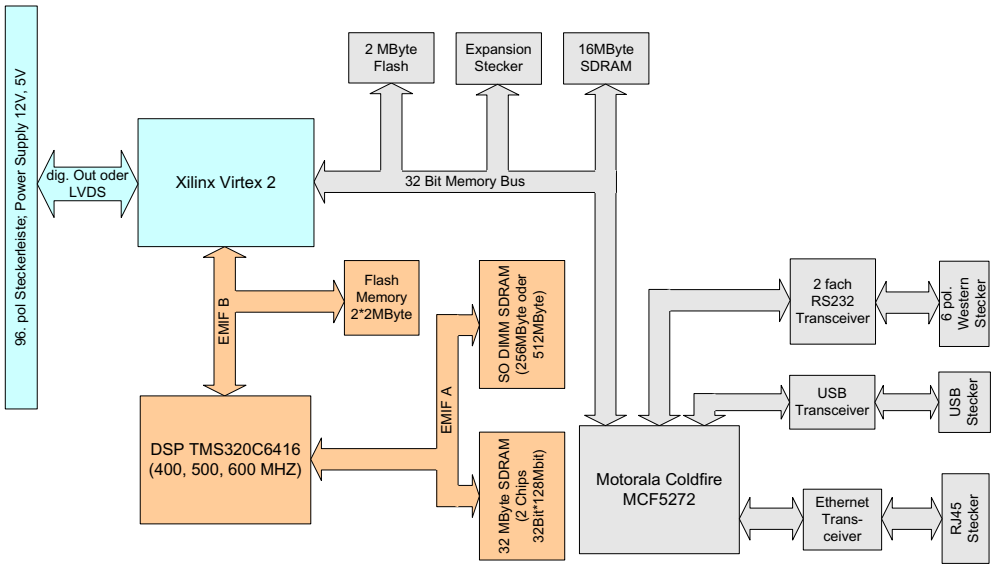

Figure 1: Block diagram of the RTS-DSP board.

want to migrate portions of the design into hardware, which constitutes a characteristic case of operation for our rapid prototyping system. Hence it is intended to replace a specific Simulink block, specified in some high level methodology, e.g. a subsystem of library blocks. Fig. 2 illustrates such a

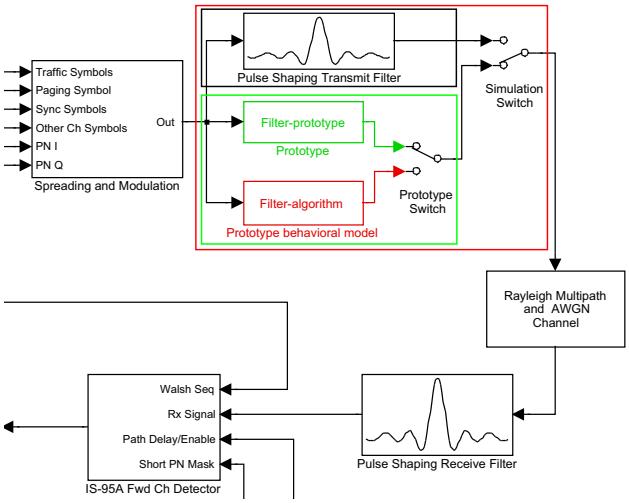

Figure 2: Simulink model expanded with prototyping blocks (in the rectangle).

scenario in the wireless field. The figure shows a clipping of a Simulink receiver transmitter model including a modulation, pulse shaping transmit, channel, pulse shaping receive and channel detect block. Now focus on the rectangle in the upper-right corner of the figure, which contains the pulse shaping transmit block, modified for rapid prototyping. The upper sub-rectangle includes the original filter; the lower sub-rectangle includes two rapid prototyping blocks. The block named `Prototype behavioral model` is a software model of the prototype, solely executed on the PC. Before one attempts to translate a prototype into hardware it is preferable to verify the behavioral model. The `Prototype switch` assures that only the output of the selected block is used in the model. Once co-simulation of the behavioral model produces satisfying results the hardware prototype can be generated and eventually co-simulated too, after the `Prototype switch` had been actuated. To verify the functionality of the prototype blocks the (saved) results of simulation runs with different switch positions can be compared. Another approach would be to build the difference of the original pulse shape filter output and its respective prototyping block output by means of an addition block, which would allow for co-simulation in a single run.

3.1 Prototyping Input

As was demonstrated in the previous paragraph, in a top level view, the purpose of the prototyping system is to build a prototype represented by a single Simulink block.

Several approaches exist of how to incorporate hardware into a Simulink model. One technique is to utilize the structure of a model and convert it into a corresponding HDL description, whereas each block has its own, generic HDL description. Structural information in the model is used to interconnect the HDL modules. This approach allows the use of even extremely optimized or pre-synthesized cores, since only the interconnections for the modules are created and their generics actual values are replaced. A successful implementation of this concept and probably the most popular among these tools is Xilinx System Generator [10]. Though

being very promising this approach suffers from missing means of control, since the model only allows to access the interconnections and not the block internals. Building sophisticated control logic with the provided library blocks is very labor intensive and will, due to a required workaround, often result in suboptimal results. Nevertheless this is a comfortable solution suitable for users requiring only standard components.

Another approach, providing more *flexibility* and better access to the algorithms *internals*, is to specify the input to the prototyping system in a high level programming language. Such language must be widely accepted to avoid the tedious acquisition phase of a new language, suggesting to use C/C++. Following the "one source" paradigm in the five ones approach of [2] we decided to set up the entire rapid prototyping flow on a single source[4]. This implies that all block information, in addition to the algorithm relevant to the simulator engine, must be contained in this source. C/C++, however, does not provide facilities for specifying Simulink parameters.

But GenericC, defined by COSSAP, a well known C dialect in EDA industry , offers the specification of input ports, output ports, their associated data rates and types by introduction of a header with new language constructs. Furthermore, it was designed to describe the behavior of systems (with states), which made it a solution tailored to our purpose. Additional information, as optimization options for synthesis or selected interfaces for data exchange, are not contained in the source code, but may be specified at a later time as tool options in the graphical user interface. The source only contains information which is directly required for simulation.

3.2 Simulation Semantics

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. Simulink supports continuous time systems as well as discrete time systems. Since our rapid prototyping system focuses on digital hardware only we do not have to deal with Simulink's continuous time features. For discrete time systems an appropriate solver exists, which guarantees the following:

- The 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 simulating under Simulink some issues have to be considered, as explained in the following.

3.2.1 Issues

In addition to supporting single rate systems our prototyping system 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.

Simulations with a high count of blocks (several hundred) and deep hierarchies may cause a severe performance decrease due to the overhead caused by the scheduling algorithm. When the simulation additionally is very exhaustive it may be desirable to reduce the overall simulation 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 is only invoked once by the scheduler.

3.2.2 GenericC

We decided to use GenericC as source language, because it allows to tackle the above mentioned problems by means of the language integrated features. Thus a data rate can be specified for each input and output port which, according to our simulation semantics, corresponds to the block processing size. But 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. Designers, who code blocks in GenericC, have full control over the handling of sample rates because it takes place in the GenericC code. Any port is mapped into an array with a size given by the data rate. Full control means that any value of the array, semantic the port value at the time provided by the array index, can be accessed (via the index operator `[]`). The user has to take care of the processing of the entire block in this case. It would have been possible to restrict the users data access rights and perform the iteration over the array automatically to exclude potential human errors. But 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 we decided to let the user take care of the port access. Section 4.2 provides source code illustrating the actual port access methodology.

3.3 Hardware Generation

This section provides the key concept of FPGA prototype generation, which forms the basis of our rapid prototyping system.

3.3.1 C to VHDL

Before a prototype can be implemented on FPGAs, it must exist as HDL code written in RTL style, such that it is possible to perform hardware synthesis with common tools (e.g. Leonardo Spectrum [11]). To maintain automation in our rapid prototyping system the prototype must be *generated* from GenericC code. Ideally, the prototype should exist as entity architecture pair to ease integration into other VHDL code. Fortunately, GenericC comes with language constructs, in particular `INPUT PORT` and `OUTPUT PORT`, which allow for direct mapping into a VHDL entity.

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

But a widely accepted hardware semantics of ANSI C/C++ (*not* SystemC) is still not defined. Writing a program in C is a straight forward task, but describing a piece of hardware is much more difficult because a C program is a *sequence* of operations, one after the other, while in hardware as ASICs or FPGAs operations are often 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 HandelC [12] uses `par { }` to tell the compiler about parallelism. Although this approach is successful[5] 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 VHDL translation tool or it is denoted implicitly in the code by language compliant constructs.

VHDL code, suitable for synthesis, has to be coded in specific patterns to allow *hardware inference*, i.e. the synthesis tool allocates hardware on the FPGA for this pattern. In general the hardware used for inference is all-purpose hardware, applicable for different tasks, like look-up tables or flip-flops. Additionally most FPGAs provide optimized hardware blocks which are dedicated for 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.

Since creating a C to RTL tool would have been too much effort, we decided on using an external tool. Among various alternatives[6] we have selected A|RT Builder even though it is no longer commercially available because it does not use language extensions (but compiler directives), comes with a fixed-point library for hardware and software and is suited for proof of concept of our rapid prototyping system.

A|RT Builder generates a register template (more precisely a sequential process) for each static C variable it finds. The sequence of operations in the C algorithm itself is translated into a sequence of accordingly mapped VHDL operations in a combinatorial process, which usually results in high FPGA area consumption when synthesis is performed. E.g., loops are unrolled such that each iteration results in a separate hardware resource (maximum parallelism). By using specific coding styles it is possible to share resources over iterations or blocks.

3.3.2 VHDL to FPGA

Once the VHDL prototype has been generated it must be possible to access it from the "outside world" to charge it with simulation data. The "outside world" constitutes in our case an existing outermost or top VHDL shell, provided by ARCS. This top shell provides the CPU bus (see Section 2), which transmits simulation data. To provide simulation data for more than one bus cycle it must be stored in some kind of memory, which constitutes an interface between top shell and prototype. This IF, which is called 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 port of the prototype can have its own data type, always being an assembly of bits, and its own data rate, which will result in different memory sizes. To keep the middleware[7] 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.

When the middleware finished data transmission for the input ports, the prototype is enabled by setting a control register in the FPGA which then will begin computation. Depending on the number of iterations, scheduled in the GenericC 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 causes the board CPU to upload the prototype's results to the PC.

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. But our prototype is not used in a real time system because there are no constraints on how long a co-simulation may take. Hence 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 PLL and can be adjusted in a broad range.

3.4 Design Flow

Based on the theoretical insights attained in the previous sections a chain of tools can be established, which is described in this section.

Fig. 3 illustrates the design flow we have implemented in our tool. The flow begins in the upper left corner with the GenericC source. This source is analyzed by our parsing tool which saves its results in a database file. Then the flow splits up in two branches:

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

In the first branch the database is read and the prototype module is generated, which is compiled together with the S-Functions (S-Functions are user defined Simulink blocks) into the respective Simulink Blocks. Hardware generation is performed by creating the hardware C source for A|RT Builder, converting it into the VHDL prototype and synthesizing it together with the generated wrapper to an FPGA configuration file.

3.4.1 Input Analysis

To extract all information from the GenericC input file, the parser GCParser has been written (see Fig. 3). Since it must not be sensitive on white spaces the source is firstly processed by a scanner, which also discards all C/C++ comments (line comments `//` and section comments `/* ...*/`). Its output to the parser is a stream of tokens. Language extensions of GenericC are used to create a data structure which contains information about the prototypes IF: input/output port numbers, types, names, rates and state type,

[5] HandelC tools are already over 10 years on the market.

[6] A|RT Builder (sold to ARM in 2003) from Adelante Technologies, CoDeveloper from ImpulseC [13], DK Design Suite from Celoxica [12] and CatapultC from MentorGraphics [14].

[7] μCLinux adapted for Motorola's Coldfire is used to handle communication and peripheral tasks [15].

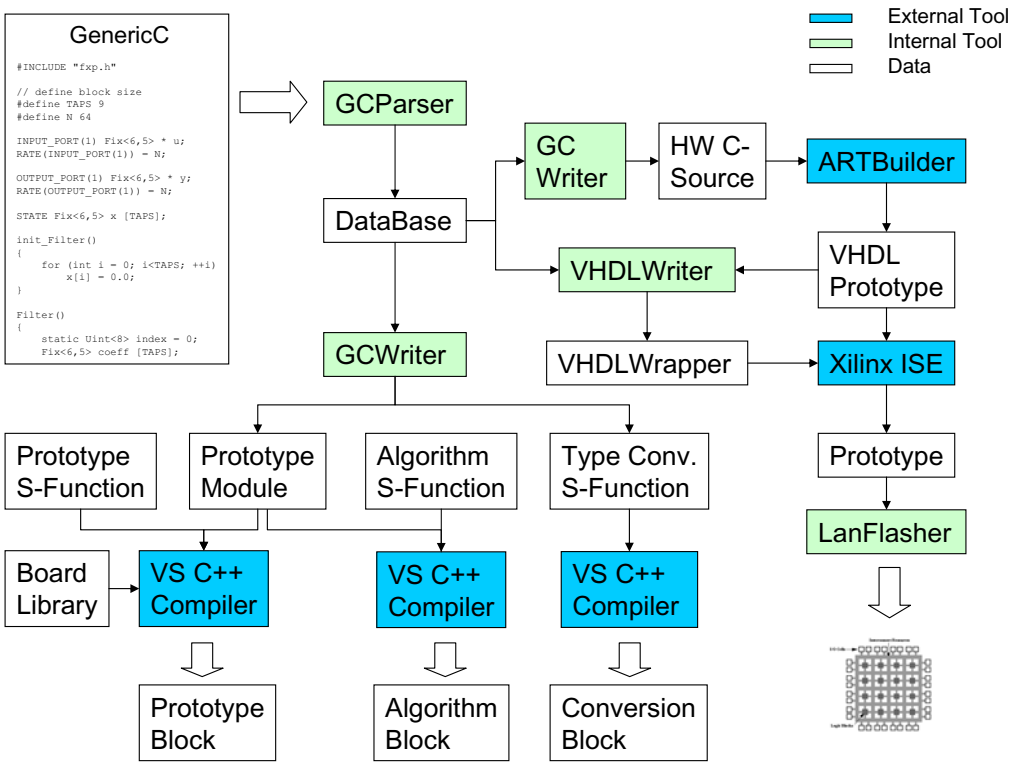

Figure 3: Rapid prototyping design flow

name and rate. Additional to ANSI C types fixed point types, defined in the A|RT library, are supported. By means of this type, variables of arbitrary length up to 32 bits may be used, whereas the fractional part's length can also range from 0 to 32. A valid GenericC source must include two functions:

Initialization The initialization function (must be named `init_Prototype`) is called only once at the simulation startup and should initialize the state variable, which must have been previously declared.

Algorithm The algorithm function (must be named `Prototype`) describes the behavior of the block.

Function contents are not analyzed by the parser but saved in the database file together with the prototype IF information mentioned above. It is necessary to save the parser's output since the tool VHDLWriter requires its information, but cannot be invoked together with (or immediately after) the parser. GCParser takes only one source file which is the same for the entire rapid prototyping design flow.

3.4.2 Simulink Block Generation

The first branch "Simulink code generation and compilation" in Fig. 3 illustrates what tools and sources are required to generate Simulink blocks. The most important role plays the C++ code generator GCWriter. It reads the database file and generates the according C++ prototype module which consists of a header and an implementation file. All information from the GenericC file, such as ports and states, is encapsulated in this module, which offers its functionality in form of a class.

S-Functions are generated from generic S-Function templates, which fit to all supported data types and port sizes.

Algorithm S-Function The prototype module contains a prototype class, which forms the *behavioral* model of the prototype. The algorithm S-Function creates an object of this prototype class. When prototype specific functionality needs to be accessed the corresponding class methods are invoked.

Prototype S-Function Unlike the algorithm S-Function the prototype S-Function does *not* execute behavioral code. Its task is solely the access to the FPGA prototype. Nevertheless the prototype module is required since it contains functions which convert simulation data to binary data and vice versa. When all input port data has been converted to binary format, board library functions send it via ethernet (see Section 2.4) to the RTS DSP board. Once the board completed its computations and all results are received they are converted back to Simulink values (double float format).

Type Conversion S-Function This S-Function is used to convert Simulink data to the supported fixed-point numbers in order to investigate truncation and overflow behavior. However their usage in a Simulink model is optional, since a conversion to fixed-point numbers is performed implicitly in the co-simulation blocks.

Once GCWriter has generated all S-Functions, the Visual Studio C++ compiler generates MEX (Matlab executable) DLLs which can be invoked from Simulink.

3.4.3 *Hardware Generation*

The second branch in Fig. 3 displays a set of tools used to generate a hardware prototype and configure the FPGA.

First the hardware C source is generated by GCWriter (as defined in Section 3.4.2). GCWriter reads the database and uses its interface information to generate the corresponding inputs and outputs for A|RT Builder. The algorithm itself is pasted in its original version into the hardware C source, except that GCWriter adds a system flag which is used to determine the termination of the algorithm. This guarantees maximum flexibility to the designer but requires some coding styles which are illustrated in Section 4. Furthermore, detailed knowledge of A|RT Builder's translation methodology is obligatory, when designers want to optimize hardware results for specific parameters like area consumption. E.g., it is possible to pipeline (divide critical paths) a design by using static variables.

A|RT Builder translates the hardware C source into the VHDL prototype, which is analyzed by VHDLWriter in conjunction with the database. Actually, it should be sufficient to read the database but it lacks A|RT Builder's VHDL type information. This type information can only be obtained from the VHDL prototype, since it uses these types. Once VHDLWriter's IF information is complete, it generates the wrapper, which consists of registers. Although registers are costly in terms of area consumption, we decided to use them because they provide access to *all* stored data in each clock cycle contrary to RAMs which provide only one word at a time.

The top shell, together with the embedded wrapper and prototype, is synthesized by Xilinx' ISE and further implemented as .bit file. All implementation constraints and other required sources are taken from a template project. Finally the FPGA is reconfigured without a board reset by LanFlasher, which loads the .bit file into the SDRAM (see Section 2.1) and activates the DSP to update the FPGA.

3.5 User Interface

The rapid prototyping system is operated via a graphical user interface, which is described in this section.

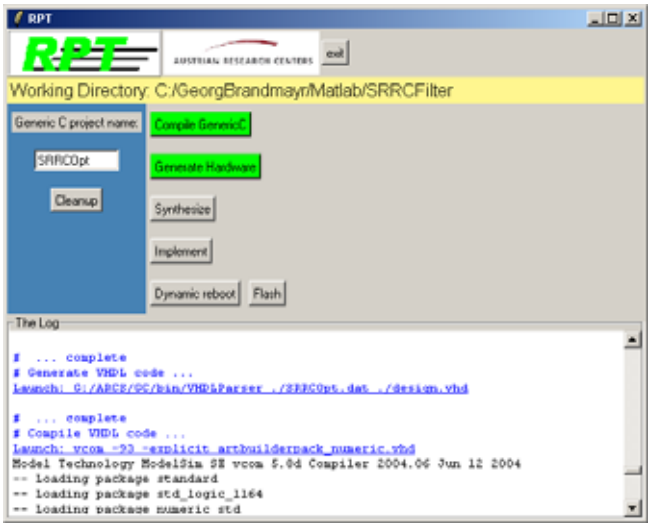

Figure 4: Graphical User Interface

Our prototyping system and all its integrated sub tools as described in Fig. 3 are designed to operate in arbitrary working directories. Hence users typically choose their model's directory since all S-Functions are required to be in this directory and our tool creates all final output data in its working directory (WD). A Microsoft Windows shortcut might be used to start our tool, which displays its WD in the top section. Fig. 4 shows the graphical user interface, which expects the user to enter a GenericC file name. Once the file has been found, all buttons are enabled and serve the following purposes:

Compile GenericC allows to create behavioral prototype from the GenericC source. GCParser analyzes it and saves results in the appropriately named 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 and including the prototype module. Refer to the left branch of the design flow tree, starting at the "database" block, in Fig. 3.

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. Refer to the right branch of the design flow tree in Fig. 3.

Synthesize invokes XST (Xilinx Synthesis Tool) to synthesize the VHDL design and output a net list.

Implement The net list is translated, mapped, placed and routed in the given order by the corresponding Xilinx tools, which results in an FPGA configuration file.

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 of invoked tools is logged in the window at the GUI's bottom. Warnings and errors are highlighted yellow and red respectively to allow for easy navigation in large logs.

4. DESIGN EXAMPLE: SRRC FILTER

The capabilities of our rapid prototyping system are demonstrated by an example which is typical for the wireless field: an SRRC transmit filter. It is used to limit the required bandwidth of the transmitted symbols by shaping them with a finite pulse form. Fig. 5 illustrates the general FIR filter implementation (tapped delay line) with the order M, which is described by

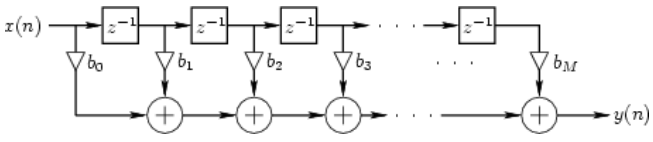

Figure 5: General finite-impulse-response (FIR) digital filter

$$y(n) = b_0x(n) + b_1x(n-1) + \cdots + b_Mx(n-M) \quad (1)$$

$$= \sum_{m=0}^{M} b_m x(n-m) \quad (2)$$

According to (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.

4.1 Specification

The filter is specified by the following parameters:

- 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 order

$$M = 2 \cdot K \cdot T_g \quad (3)$$

which results in 13 coefficients for our example.

4.2 GenericC source

A direct implementation of Fig. 5 would lead to a filter requiring M unit delays and $M+1$ multiply-accumulates, which gives according to the specification twelve registers and thirteen multiplications. Although this should fit easily into the FPGA (see Section 2.1), even if implemented fully parallel, the computational effort can be decreased by using mathematical optimizations. A closer look at SRRC filters unveils that their impulse response is symmetric ($b_n = b_{M-n}$) and its number of coefficients $M+1$ is always odd due to (3). Symmetry allows to first add samples of identic coefficients and then build the product. Upsampling requires the insertion of zeros into the input signal. Since a product of zero and any coefficient will be zero this implies that just every K^{th} multiplication has to be performed. Additionally, since just 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 bits 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. GenericC's 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 ($b_1, b_3, b_5 \ldots$) reference here to the coefficient indices, whereas duplicate coefficients have been omitted. The variable `index` is used to iterate over the port arrays; `even` indicates whether the even coefficients are to be used.

```
    // 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 is executed only each second iteration since each second input value is a zero. Coefficient multiplexing (the `if` in the body) and symmetry utilization allow to "shrink" the MAC section.

```
    ++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 can be done in one invocation, this would result in substantially larger HW prototypes.

4.3 Hardware Results

Once the functionality of the behavioral prototype has been verified, the HW prototype can be generated. Created VHDL sources are hardly human readable, however 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%

:

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
```

4.4 Co-Simulation

Co-simulation verifies the generated behavioral and hardware prototype. Of course the filter prototypes could be used directly in the model of an application like in Fig. 2, but long simulation runs suggest to focus on the prototypes themselves. Once their outputs had been verified they can be "trusted" and used in applications. In Fig. 6 all filter representations are driven (the behavioral prototype, hardware prototype and Simulink reference block) with a Dirac pulse sequence. Hence the filters impulse responses can be seen at their outputs in Fig. 7, which are, except for quantization errors, equal.

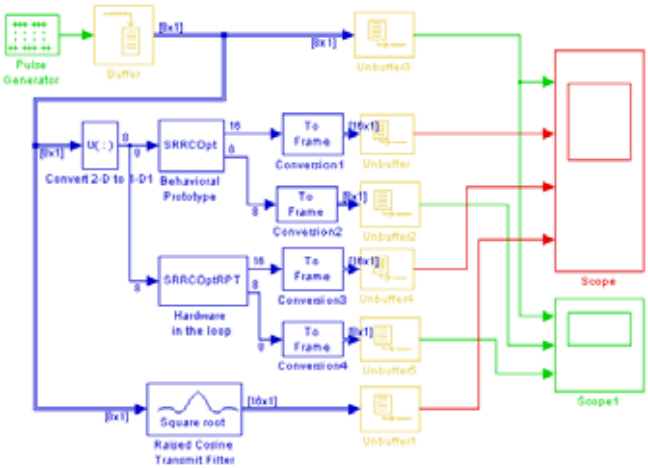

Figure 6: Behavioral prototype and Hardware in the loop co-simulation.

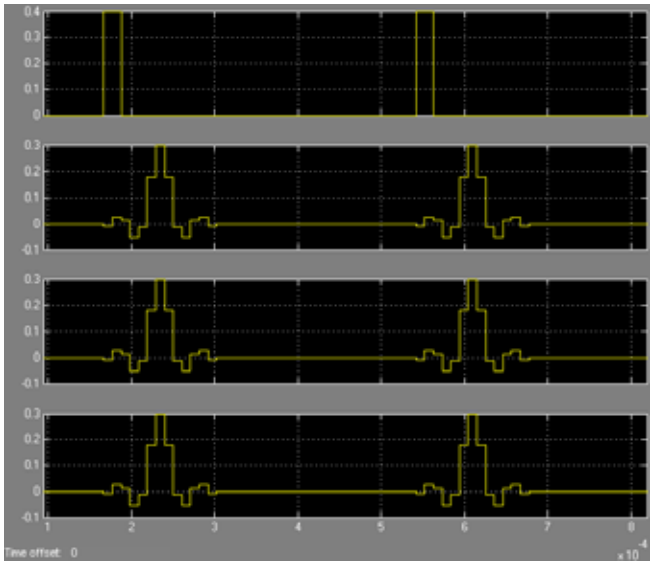

Figure 7: Upsampled filter output of "Scope" in Fig. 6.

5. 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 ours [2, 9, 16]. However, our design flow was implemented as a real, executable *tool* what emphasizes its applicability in a product. Our tool requires no further input, except tool options, since all source code is *generated*. Implementation times for prototypes are substantially reduced, e.g., the filter presented in Section 4 was implemented in seven hours including concept, coding and tool application.

The EDA industry is 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. The introduction of ANSI C/C++ as new abstraction layer is the most common approach (for examples refer to Section 3.3.1). But this methodology has not been widely accepted (not SystemC) in the EDA industry, since the resulting hardware quality is not sufficient. If a designer considers our tool for product design, the following should be considered.

C++ to VHDL translation is a crucial part in our 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 higher resource consumption. Hence this methodology is not applicable for highly optimized ASIC designs, produced in large quantities, since using less 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 GenericC code, which is required to make the design fit into the given FPGA; or atomic C++ instructions may not be split up to optimize critical paths. These problems still must be solved by providing manually optimized

VHDL designs, which proves that our approach, with *current* C to RTL conversion tools, is not applicable for building products.

However, designers, unfamiliar to VHDL, can use our tool to create HW prototypes. Although results are not optimal, a prototype can prove a design's feasibility and exhibit some of a final product's features as discussed in Section 1. Furthermore, Simulink enables to integrate prototypes into complex models, which allows for comfortable design verification at a very high system level.

The above mentioned constraints are hints for future optimizations. Clearly, C to VHDL conversation has to be improved. CatapultC from MentorGraphics [14] seems to be a very promising tool to replace A|RT Builder. Another option, avoiding CatapultC's high license cost, is ImpulseC's CoDeveloper [13], which uses C to VHDL conversion technology from Stanford University. But besides C to VHDL conversion, there is additional space for improvements. It would be useful to use Simulink's integrated fixed-point types as prototype I/O, since they are widely supported (e.g. MentorGraphics "Link for ModelSim" uses this types). Finally, a very promising approach is to *unite structural* and *behavioral* synthesis in our tool, i.e., to use the structure of a Simulink model (compare Xilinx System Generator [10]) for interconnection of several GenericC blocks and subsequently VHDL entities.

6. ACKNOWLEDGEMENTS

Thanks to Roland Lieger for his support with LanFlasher and the boards firmware, Florian Schupfer for all help regarding his FPGA framework as well as Martin Holzer who gave us great support with A|RT Builder. Also thanks to Peter Brunmayr for carefully proofreading this paper.

REFERENCES

[1] GSMWorld, "One million customers milestone," http://www.gsmworld.com/about/history/index.shtml, 2004.

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

[3] C. Velasquez, M. Gauckler, and M. Polasek, "Integrated FPGA rapid prototyping environment," Master's thesis, ETH Zürich, July 2004.

[4] 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.

[5] G. Meindl-Pfeiffer, R. Kloibhofer, F. Kaltenberger, and G. Humer, "Multistandard development platform for MIMO software defined radio," in *EUSIPCO*, Antalya, Turkey, Sept. 2005.

[6] F. Kaltenberger, 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.

[7] R. Kloibhofer and H. Eberl, "Kanalsimulator v1.0 2003-11-10," ARC Seibersdorf research GmbH., Vienna, Austria, Tech. Rep., 2003.

[8] G. Humer, H. Ritt, and M. Meyenburg, "Antenne mit Intelligenz," in *Embedded World Kongress 2003*, Nürnberg, Germany, Feb. 2003.

[9] A. Adjoudani, 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, 2003, pp. 440–451.

[10] Xilinx, "System generator extended feature set," http://www.xilinx.com/products/software/sysgen/features.htm, 2005.

[11] MentorGraphics, "Leonardo Spectrum," http://www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum/, 2005.

[12] Celoxica, "Handel-C," http://www.celoxica.com/technology/c_design/handel-c.asp, 2005.

[13] ImpulseC, "From C to FPGA," http://www.impulsec.com/C_to_fpga.htm, 2005.

[14] MentorGraphics, "Catapult-C synthesis," http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm, 2005.

[15] S. Krzalic, "μCLinux for Motorola Coldfire," Master's thesis, FH Technikum Wien, July 2004.

[16] A. Burg, 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*, Feb. 2002, pp. 8_1–8_6.