Custom Hardware Integration into DBT-based Processor Simulation

Steffen Köhler and Rainer G. Spallek
Institute of Computer Engineering
Technische Universität Dresden
01062 Dresden, Germany
Email: steffen.koehler, rainer.spallek@tu-dresden.de

***Abstract*—High performance simulation of processor architectures at instruction set / behavioral level often utilizes Dynamic Binary Translation (DBT) techniques to achieve an efficient mapping to the simulation host. While the behavior of most standard processor operations can be directly translated into host processor instructions due to their similarities, the behavior of complex application specific instructions or peripheral components are less suitable for host processor execution. In this paper we discuss the migration of application specific behavioral processor model partitions to field programmable accelerator hardware to achieve an overall co-simulation speedup. For an in-depth evaluation, the integration of an off-the-shelf Field Programmamble Gate Array (FPGA) into our DBT-based processor simulation framework RUBICS (Retargetable Universal Binary Instruction Conversion Simulator) was considered. RUBICS is a flexible behavioral modelling and simulation platform framework for embedded processor architectures and complete Systems-on-a-Chip (SoC). In a case study, we show the behavior model migration of an application specific peripheral co-processor into a synthesizable hardware description mapped to the FPGA accelerator. Only minor changes are required to the original ARMv7 processor model description given in RUBICS's dedicated Architecture Description Language (ADL). An overall simulation speedup between 300% and 540% has been achieved by migrating the main calculation partition of a numeric transform peripheral to the FPGA accelerator. Challenges of the communication-driven model partitioning as well as the achievable simulation speedup are discussed.**

***Index Terms*—Processor Simulation; Dynamic Binary Translation; FPGA Accelerator.**

I. Introduction

The behavioral simulation of embedded processor cores is essential for a successful design of System-on-a-Chip (SoC) architectures. Beside the processor core behavior, a detailed analysis of interaction and communication between processor core and peripheral/dedicated components plays an important role in the SoC test/verification process.
The current complexity of hardware- and software-components requires an analysis of the SoC as a whole at an early stage of the design process. Contrary to performance-oriented approaches that focus on peripherally observable behavior (emulation), the behavioral simulation additionally has to provide a short modeling turn-around cycle as well as the demanded observability of the relevant model state. To achieve this, a behavioral processor simulator should provide rapid modeling capabilities at a high abstraction level (instruction set, instruction behavior, IO behavior), visibility of internal flow (registers, data path), as well as a high simulation speed. As an increased state visibility introduces additional modelling, execution and data recording effort, the simulation model has to be carefully adapted to the particular observation requirements. This can be achieved e.g. by injecting selective debug operations into the behavioral model specification.
Beside the early design stage requirements the simulation environment could also be used to reconstruct the execution profile of embedded software modules. Through architecture model instrumentation, a detailed control- and data-flow trace may be obtained for an in-depth software analysis.
The most essential part of the processor simulation is a model description that provides information for equivalent mapping the target instruction flow to the simulation host. The quality of the translation process and the resulting fitness to the host processor architecture determines the achievable simulation performance. Common methods used for high speed behavior level processor simulation utilize binary translation techniques [1]. The trade-off between translation effort amortization and available performance can be effectively improved by applying just-in-time (JIT) compilation, where target instruction sequences are dynamically mapped to the host processor at runtime. Common approaches of JIT-based simulators and emulators use widely available runtime environments, such as Java Virtual Machine (JVM) [2], Common Language Runtime (CLR) [3] or PyPy [4] as well as dedicated compilers like Tiny Code Generator (TCG) [5]. As a matter of fact, the translation quality essentially depends on the degree of similarity of both target and host processor architectures. This problem especially involves the mapping of dedicated custom target instruction set extensions or peripheral models. It can be overcome by supplying a more flexible mapping platform on the simulation host, thus allowing a higher degree of adaption as a result of the translation. A promising platform extension approach is the integration of FPGA hardware in conjunction with a partitioning of the simulation model.
In this work, we focus on static translation of the FPGA partition, although it might also be possible to benefit from just-in-time utilization of the FPGA.
Among different available simulation environments, we have chosen the RUBICS [6] platform framework, which utilizes the open ECMA International Common Language Runtime (CLR) [7] standard platform. Advanced retargetability and modeling capabilities are provided through a dedicated architecture description language (ADL), whereas extensibility is granted by the underlying CLR language support (C#, VB, etc.). External component libraries (DLL) facilitate the integration of complex peripheral model descriptions as well as FPGA-

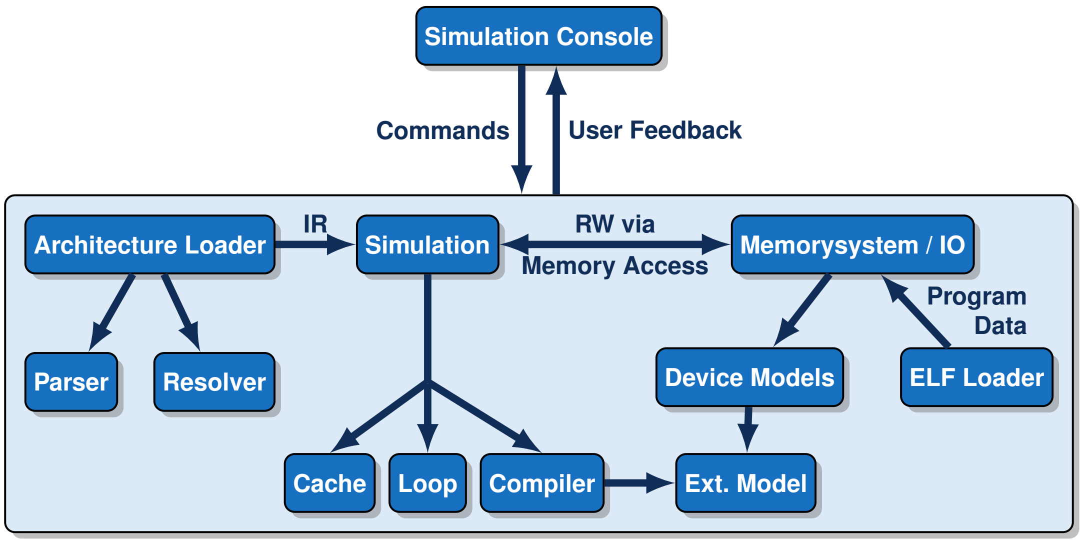

Figure 1. RUBICS Platform Framework

based model partitions into the ADL description without any changes to the simulation core.
In this paper, sections II and III introduce the RUBICS simulation platform and the provided processor architecture description methodology. Section IV outlines the binary translation based simulation process. A detailed description of custom hardware mapping to an FPGA accelerator and its performance evaluation is given in sections V and VI respectively. Finally, a summarized discussion and generalization of the achieved results is given in section VII.

II. Simulation Platform

The chosen simulation platform framework RUBICS provides the underlying infrastructure for high-performance processor simulation. It allows both structural and behavioral architecture description in a dedicated architecture description language (ADL) and supports embedding external core or peripheral models. Enhanced test and debug capabilities enable versatile control/observation of the simulation process. The framework is composed around a core component, that can be dynamically extended by loading simulation model or support libraries at runtime. Any Common Language Runtime (CLR) [7] compatible module can be made available to the RUBICS platform framework. Beside the behavioral model description in a supported language the CLR, it also grants direct access to the native system resources of the simulation host. This significantly simplifies both the integration of custom hardware models into the core model as well as extending the simulation host by additional FPGA-hardware. The CLR furthermore implements a dynamic execution interface to host processor which complies to the Common Intermediate Language (CIL) bytecode specification. The versatile bytecode interface is not only used for CLR module representation, but is also target by the dynamic binary translation process. Figure 1 gives an overview of the basic structure of the RUBICS framework.

The integration of FPGA hardware into the simulation host and its application as processor simulation accelerator is generally supported by the RUBICS platform framework in two different variations.

- Loose coupling by a bus interface for peripheral or co-processor models
- Tight coupling by a plugin interface for direct access from the ADL behavioral model

Although both possibilities could have been considered for further investigations, we focus on the bus interface integration for partitioning reasons. The plugin access would require advanced low latency communication properties, which the available interface of the chosen FPGA hardware platform unfortunately could not offer.

III. Architecture Model

The main concept of a retargetetable processor simulator by mean of combining a generic infrastructure with a dedicated architecture description language (ADL) was already suggested in [2]. Contrary to a programming language or hardware description language, an ADL model can be specified using a common structural template. The remaining description work only requires the specification of unique structure and behavior, such as processor state, instruction set, and instruction behavior. The architecture description is composed of different sections:

- `import` References to CLR library models,
- `bus` Bus-oriented loosely coupled devices
- `plugins` Tightly coupled plugins,
- `context`, Context state variables,
- `decoder`, Instruction decoder description,
- `behavior`, Instruction behavior description,
- `interrupts`, Interrupt description.

```
bus mem {
    unit = 8;
    endian = LITTLE;
    access {
        byte = 1;
        short = 2;
        word = 4;
    }
    devices {
        ram mem {
            base = 0;
            size = 0x40000000;
        }
        fpga fpga_dev {
            base = 0x40000000;
            size = 0x10000000;
        }
    }
}
plugins {
    plugin.fpga fpga_plug;
}
context {
    uint pc;
}
decoder {
    fetch {
        bus = mem;
        pc;
    }
    context {
        uint instruction_word;
    }
    operation oper {
        instruction_word = fetch.word;
        IPC;
    }
}
behavior {
    operation IPC {
        fpga_plug.ExecFunc();
        pc += 4;
    }
}
```

Figure 2. Architecture Description Language

Figure 2 shows a simplified architecture model with both tightly and loosely coupled model components mapped to an FPGA accelerator. The description consists of structural and behavioral specifications. For demonstration reasons, the instruction behavior only includes a program counter increment followed by a plugin function `ExecFunc()` invocation.

The declaration of the memory bus `mem` not only describes the main memory and peripheral mappings, but also specifies an embedded FPGA based accelerator device `fpga_dev`. Detailed access pattern sizes and alignment hints are given in the `access` section, that makes the bus interface available to the behavioral operations as named expression, e.g. `mem.byte[<address>]`. Through the `unit` attribute the data granularity (smallest addressable data unit) can be set.

Although the general execution flow is implicitly determined by the RUBICS core, the `fetch` environment has to provide the information about the proper instruction memory interface (`mem`) and the fetch address (`pc`) of the next instruction. The decoder may maintain its own global context `context`, which is especially favorable when decoding complex instruction sets.

The ADL makes the standard CLR data types available to the decoder and instruction behavior descriptions. Unique type conversions will be done automatically. A huge selection of support functions is available for specifying dedicated bit-level and floating point operations. Fixed point literals are handled with arbitrary length upon its use in a particular variable operation or assignment. This significantly simplifies constant propagation and reduces the number of required range checks. The architecture description will be translated into an internal Intermediate Representation (IR), which holds the information needed for the binary translation and the JIT compilation process.

IV. Binary Translation

The binary translation is the main task prior execution on the simulation host. After the target architecture and the application binary have been loaded and the Program Counter (PC) has been set to the appropriate start address, execution flow is transferred to the main simulation loop. According to the decoder specification in the IR, the instruction stream gets decoded and behavioral IR operation are issued and executed until the flow hits a break condition. A simplified simulation flow is outlined in Figure 3.

As behavioral operations are tied directly to particular address ranges, they can be cached to avoid redundant decode operations. In case of a cache-miss, a decode operation is invoked, which stores a new behavioral block to the simulation cache. The decode operation can be avoided in case of a cache-hit and the behavioral operation block is directly available for execution. Only instructions on consecutive addresses can be mapped to a single cache entry. The size of the behavioral operation block (dynamic block) is limited by application binary control flow transfer instructions (e.g. branches). Special handling of non-contiguous control flow is reflected by the `exit` keyword (end of dynamic block) inside the decoder specification. A sufficiently high dynamic block size is vital of a low cache lookup rate and thus for a high simulation performance.

In a first step, the binary translation creates a sequence of instruction tailored copies of the behavioral IR, which will then be optimized to eliminate redundant operations and variable access. The additional optimization effort (flow analysis) can easily be amortized, except for very short simulation runs. During the translation of the IR, native interface and plugin invocations are directly handled using native CLR calls for low overhead. A lookup-based memory delegator offers low latency access to multi-device bus interfaces with nearly constant time aperture access. Target memory blocks are directly mapped to virtual user address space of the simulation host by operating system functions utilizing copy-on-write pages.

Furthermore, hot-spot compilation [8] is applied to the simulation process, which delays the translation of the behavioral IR copies into CIL bytecode until they have reached a certain execution count threshold. The final translation of the CIL bytecode into host machine instructions is transparently maintained by the CLR-internal JIT compiler.

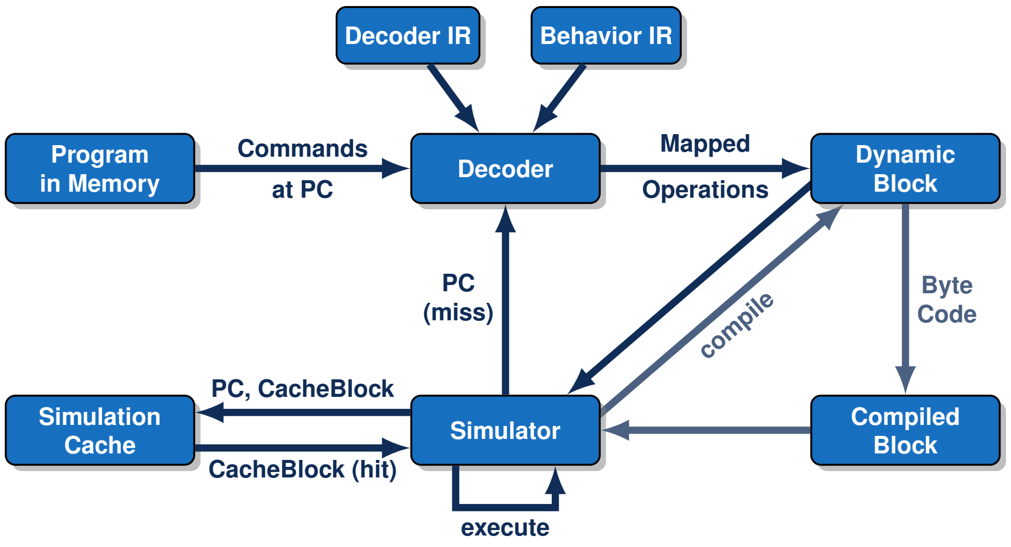

Figure 3. Simulation Cycle

V. Embedding Custom Hardware

For a reasonably complete behavioral processor description, the core architecture ADL model has to be extended by additional peripheral or plugin models. This is already supported by the RUBICS platform framework by its library concept. The behavior of a loadable library component can easily be specified using any CLR compatible programming language (C#, VB, etc.). Depending on the communication pattern and resource requirements, a migration of any RUBICS library component to an FPGA accelerator is basically possible. To achieve this, a synthesizable Register Transfer Level (RTL) model partition has to be manually created and described using a Hardware Description Language (HDL). The FPGA mapping decision is driven by the availability of such a HDL description (or the effort to create this description) and the potential performance gain including the communication overhead.
As FPGAs can only be accessed by the simulation host using a limited scale of available processor interfaces, inter-partition communication overhead directly relates to the simulation host platform. Although it would be possible to consider tightly coupled processor/FPGA host platforms, the best cost/performance trade-off can be achieved by integrating an inexpensive PCIe-based FPGA-board into a PC/workstation environment. Unfortunately, PCIe-based inter-partition communication heavily depends on the available PCIe transfer profiles, thus limiting the influence on communication latency and throughput. Therefore, only a streaming-based communication approach could be considered in this work.
Figure 4 illustrates the embedding of a custom behavioral model into the simulation process using FPGA hardware.

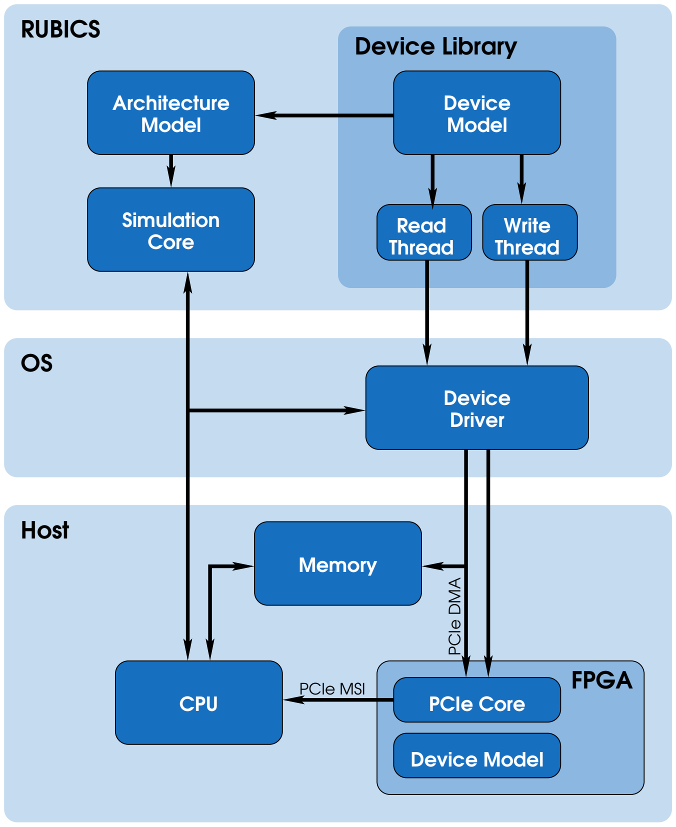

Figure 4. Simulation Environment with FPGA-Hardware

VI. Performance Analysis

In a particular case study, the FPGA hardware integration into the simulation host and its utilization as execution platform for a custom peripheral model can be demonstrated. The host system is composed of a fairly recent Intel Core-i5 6500 (SkyLake) with 16GB DDR4 memory. Through a standard PCIe expansion connector a Terasic DE5-Net FPGA board (Altera/Intel Stratix V FPGA) [9] is plugged into the main board. Only four PCIe-Lanes are used for communication. The system runs Debian Linux using an unpatched Kernel

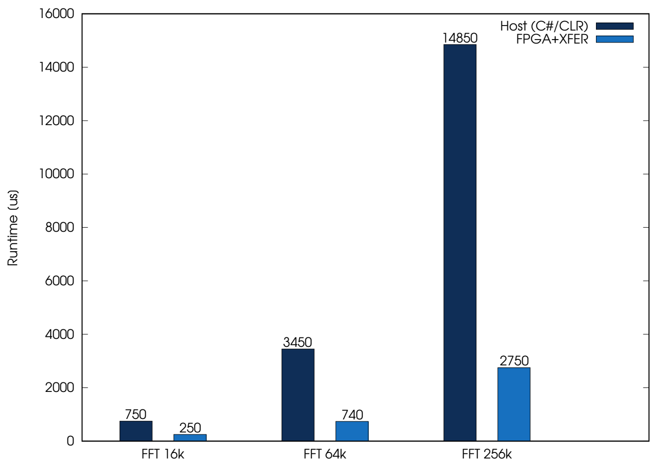

Figure 5. Simulation Runtime

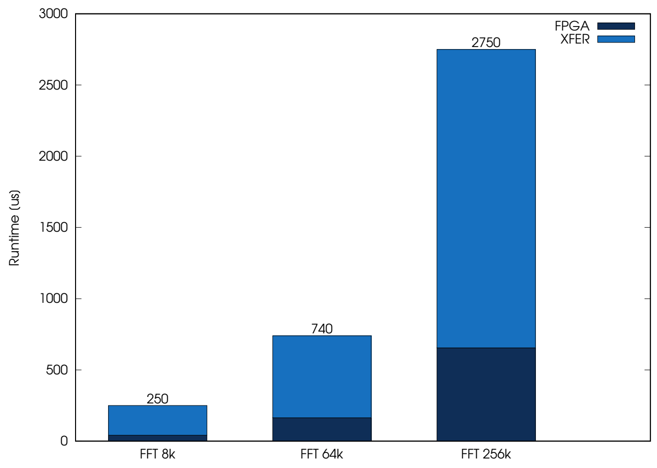

Figure 6. FPGA Calculation vs. Transfer

v4.10.1. All performance measurement were carried out on an ARMv7 Thumb2 instruction set model [10] in a RUBICS v0.6 platform framework on top of MONO CLR v4.8. The peripheral model chosen for FPGA mapping has been selected with main emphasis on implementation simplicity and interface fitness to the available PCIe IP-core. A decimation-in-frequency Fast Fourier Transform (FFT) model [11] with three different block sizes was evaluated using both a C#-based CLR thread as well as a coupled FPGA-based implementation. The description of the FPGA partition not only includes Verilog HDL RTL modules, but also structural component declarations required for automated FFT IP-core generation using QSys wizard of Quartus Prime v15.1. A XILLYBUS PCIe streaming controller IP-core [12] supplies the underlying communication infrastructure including a Linux OS device driver. Although the intended optimization goal was performance-oriented, the available FFT pipeline capabilities could not be fully utilized due to a maximum clock frequency of 250 MHz limited by the XILLYBUS PCIe-core.

Figure 5 summarizes the achievable overall simulation runtime reduction resulting from the FPGA accelerator using different FFT block sizes. A more detailed impression on particular calculation and transfer effort can be obtained from Figure 6. To neglect JIT amortization effects, the measurement results have been averaged over a total of 1000 FFT runs respectively. The overall simulation performance gain including transfer overhead reaches between 300% with an FFT block size of 16384 points and 540% with an FFT block size of 262144 points. The latter is the maximum FFT size that could be implemented using Quartus QSys wizard.

The FPGA resource consumption is dominated by the FFT component and reaches 25…30% of the available DSP and RAM blocks, whereas the remaining logic including XILLYBUS glue components utilizes only 2-4% of the chip capacity.

VII. Conclusion and Future Work

In this paper, we have shown the integration capabilities of application specific custom hardware models into the RUBICS behavioral simulation flow. An acceleration of the simulation process could be achieved by the utilization of coupled FPGA hardware on the simulation host. This approach is especially useful for complex behavioral models of peripheral components of processor-based SoC. Furthermore, the embedding of external models into the architecture description was demonstrated, which allows tight or loose coupling of external custom models to the core architecture. Through the selected migration of a custom FFT model partition to a loosely PCIe-coupled FPGA board a simulation performance improvement compared to the standalone mapping to the simulation host has been demonstrated. The achievable runtime-benefit increases with the mapping advantage of the FPGA compared to the host processor and the reduction of transfer overhead between the FPGA- and host-partitions. The selected FFT example does not fully comply these demands, thus resulting in a comparatively low performance gain. Especially the communication latency of the throughput-oriented XILLYBUS IP-core lowers the achievable overall simulation performance significantly. A definition of more realistic partitioning properties and goals would therefore be desirable. Beside the computational resource specification, this would also include precise knowledge about the request/response communication round-trip time. Also, the availability of a low-latency PCIe IP-core would be generally preferable for obtaining an increased overall integration efficiency of tightly coupled custom hardware models on FPGA. Furthermore, the communication latency can be entirely neglected by breaking the request/response scheme and avoid stalling the simulation progress while waiting for any hardware response. The case is most relevant to uni-directional fire-and-forget transfers like execution flow trace recording, which is subject of prospective investigations.

REFERENCES

[1] "High speed cpu simulation using jit binary translation," 2007, URL: http://homepages.inf.ed.ac.uk/npt/pubs/mobs-07.pdf [accessed: 2017-03-21].

[2] M. Kaufmann, M. Häsing, T. Preußer, and R. G. Spallek, "The java virtual machine in retargetable, high-performance instruction set simulation," in Proc. 9th Int'l Conf. on Principles and Practice of Programming in Java (PPPJ). ACM, 2011.

[3] P. A. Wright, "Dynamic binary translation on the .net platform," Master Thesis, Victoria University of Wellington, New Zealand, 2014.

[4] D. Lockhart, B. Ilbeyi, and C. Batten, "Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers," in IEEE Int'l. Symposium on Performance Analysis of Systems and Software (ISPASS), March 2015.

[5] F. Bellard, "Tiny code generator (tcg readme)," 2016, URL: http://wiki.qemu.org/Documentation/TCG/.

[6] S. Köhler, T. Frank, M. Häsing, and R. G. Spallek, "Rubics: Ein retargierbares framework zur modellierung von prozessorarchitekturen auf der basis von .net clr," in Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS). Fraunhofer Verlag, 2016.

[7] *ECMA-335: Common Language Infrastructure (CLI)*. ECMA International, June 2012, 6^{th} edition.

[8] A. Aho, R. Sethi, and J. Ullmann, Compiler Construction. Addison–Wesley Publishing Company, 1988.

[9] DE5-Net User Manual v1.04, 2017, URL: http://www.terasic.com.tw [accessed: 2017-03-21].

[10] ARMv7-M Architecture Reference Manual, 2010, dDI 0403D, ID021310, URL: http://www.arm.com.

[11] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," Mathematics of Computation, April 1965.

[12] "An fpga ip core for easy dma over pcie," 2017, URL: http://xillybus.com.