Development of a modular and fully-digital PCIe-based interface to Real-Time Digital Simulator

Entwicklung einer modularen und voll digitalen PCIe-basierten Schnittstelle zum digitalen Echtzeit Simulator RTDS

Steffen Vogel
Matriculation Number: 304957

Master Thesis
at
RWTH Aachen University
Faculty of Electrical Engineering and Information Technology
Institute for Automation of Complex Power Systems
Univ. Prof. Dr.-Ing. Antonello Monti

Supervisor: Marija Stevic
Dr. rer. nat. Stefan Lankes

Eidesstattliche Versicherung

Ich, Steffen Vogel (Matrikelnummer 304957), versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

Entwicklung einer modularen und voll digitalen PCIe-basierten Schnittstelle zum digitalen Echtzeit Simulator RTDS

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

Belehrung

§156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit einer Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des §158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Kurzfassung

Diese Arbeit befasst sich mit der Entwicklung digitaler Schnittstellen zur Kopplung von Echtzeitsimulatoren. Zu diesem Zweck wurde ein flexibles Framework implementiert, welches den synchronisierten Austausch von Simulationsdaten ermöglicht. Harte Echtzeitanforderungen und die damit verbundenen Anforderungen an Kommunikationslatenzen im Mikrosekundenbereich machen eine FPGA-basierte Implementierung unumgänglich. Das *VILLASfpga* genannte Framework basiert auf Xilinx's aktuellen Entwicklungsumgebung und nutzt AXI4 und AXI4-Stream Busse, um bestehende IP Komponenten zu integrieren und deren Wiederverwendbarkeit zu gewährleisten. Beispiele für solche Bausteine sind Modelle, die mit Hilfe von Xilinx's Vivado System Generator for DSP (XSG) und Vivado High-level Synthesis (HLS) Werkzeugen entwickelt wurden oder auch Schnittstellen zu kommerziellen Simulatoren von RTDS oder OPAL-RT. Des Weiteren ermöglicht eine PCIe Schnittstelle den Informationsaustausch zwischen dem Field Programmable Gate Array (FPGA) und der bereits existierenden *VILLASnode* Software. Von speziellem Interesse ist die Kopplung des Echtzeitsimulators von RTDS mit dieser Software, da sie eine Reihe neuer Anwendungsmöglichkeiten eröffnet und die Flexibilität des Simulators steigert. Sowohl die Schnittstelle zum RTDS Simulator als auch die PCI Express (PCIe) Schnittstelle zwischen FPGA und Linux-Rechner wurden im Hinblick auf ihre Echtzeitfähigkeit ausführlich getestet. Einfache Modelle auf dem FPGA und Linux-Rechner wurden verwendet, um die Erweiterbarkeit und Flexibilität des Frameworks zu demonstrieren.

Stichwörter: Echtzeit, Co-Simulation, RTDS, GTFPGA, PCIe, Schnittstelle

Abstract

This thesis focuses on the development of digital interfaces for the interconnection of Digital Real-Time Simulation (DRTS). For this purpose, a flexible framework has been implemented to facilitate the synchronized exchange of simulation data. Hard real-time requirements and low communication latencies necessitate a FPGA-based implementation. The *VILLASfpga* named framework uses Xilinx' latest FPGAs and design tools. It is built around industry standard AXI4 and AXI4-Stream interfaces to warrant reusability and integration of existing Intellectual Property (IP) components. Examples for such components are models implemented with Xilinx's XSG and HLS tools or interfaces to commercial DRTSs from RTDS or OPAL-RT. Furthermore, a PCIe interface is used to integrate the FPGA framework with the existing *VILLASnode* software. Coupling the DRTS from RTDS with this software is of special interest as it opens a variety of new applications and increases the overall flexibility of the simulator. Both the PCIe and Real-time Digital Simulator (RTDS) interfaces have been evaluated in regard to their real-time performance. Simple models on the FPGA and the Linux machine have been used to demonstrate the extensibility and performance of the framework.

Keywords: Real-time, Co-Simulation, RTDS, GTFPGA, PCIe, Interface

Contents

1 Introduction

1.1 Motivation

This master thesis introduces a FPGA-based framework for distributed co-simulation of power systems. Advances in microprocessor technology and programmable logic devices like FPGAs and specialized Digital Signal Processors (DSPs) are key enablers of digital real-time simulation. Increasing performance, programability and competitive costs of such devices are the main factors for the heterogenous range of products which are available for real-time simulation today. At the same time, interdisciplinary studies of complex systems necessitate the integration of multiple platforms. Examples are multi-physics simulations of thermal and electric systems or the simulation of mechanical torques which can be found in wind turbines or electric vehicles. As the main motivation behind real-time simulation is testing of real devices, custom requirements lead researchers to design their own specialized Power-Hardware-in-the-Loop (PHIL) setups and experiments which need to be integrated into the simulation environment [4].

But not only the simulator hardware is becoming more heterogenous and diverse. Hybrid simulations combine multi-domain simulation approaches, such as co-simulation of Electro-magnetic transient (EMT)-based solvers with solvers based on Dynamic Phasors (DPs). In this case the interface is in charge of handling the conversion between these two domains.

The ordinary approach for coupling real-time simulation equipment is using analog signals. However, this method has a couple of disadvantages: Analog signals are inherently susceptible to externally induced noise which degrades signal quality and thereby limits the maximum length of the link. In addition, this noise limits analog-digital converters in their resolution. Likewise the operational voltage ranges of the converters are limited. Last but not least, every single signal requires a dedicated cabling which gets cumbersome in cases where a large amount of signals or parameters have to be exchanged. An interface might not always be used for the interface of 3-phase busses whose ranges are known in advance. Sometimes transfer functions or other complex parameters have to be exchanged. In case the parameters may vary of a wide range or their values are unknown, analog signals can not cover them in their full precision.

Power system simulation mainly requires solving of large and sparse matrix equations and has therefore been a compute-bound problem. Gordon Moore predicted the tremendous growth of computation power for the last decades. Lately, Moore's law lost its validity due to processors hitting the so called *power-wall*. The increasing integration density and higher clock speeds are demanding more power which as a result has to be dissipated as heat. However, the amount of energy which can be dissipated is limited by physics. This restriction has been a catalyst for the development of multi-core processors, distributed systems and specialized accelerators. As a consequence, power system simulation has become a communication- instead of a compute-bound problem. Especially real-time simulation is susceptible to this bottleneck as hard deadlines must be met.

Finally, competition between vendors is an obstacle for integration. There is currently no standard for real-time co-simulation. This motivated the design of a vendor-neutral hub for real-time co-simulation. Such a central hub could act a gateway between a arbitrary number of simulation nodes (simulators, Hardware-in-the-Loop (HIL) interfaces, monitoring and control devices or software). In future, this hopefully reliefs researchers from the burden of interface design and allow them to focus on their research assignment.

Point-to-point interfaces are usually sufficient for simple setups with not more than two nodes (simulators or HIL devices). But as soon as more nodes are involved the number of possible point-to-point interfaces scales quadratic. A central interconnect can alleviate this issue by providing a common interconnect for all interfaces. Adding support for a simulator becomes easy as only a single new interface must be implemented.

1.2 Objective

First ideas for this thesis arose from the demand of an interface between RTDS's DRTS and a Linux-based x86-machine. A requirement for such an interface has been the ability to synchronize exchanged signals with the time step of the RTDS simulator. Existing communication protocols supported by RTDS do not offer this feature with the exception of a FPGA-based extension card called GTFPGA. Interfacing a commercial simulator to a generic Personal Computer (PC) and FPGA significantly simplifies development of generic interfaces and thereby opens a new range of possibilities.

Previous work implemented a flexible *soft* real-time co-simulation hub on a Linux machine with similar goals. We named this software *VILLASnode*[1]. Following this scheme, the new FPGA-based framework for *hard* real-time co-simulation is called *VILLASfpga*. Based on the GTFPGA card, it realizes a flexible interconnect for a variety of components involved in real-time simulation like DRTSs, FPGA-models or custom HIL interfaces. This framework lays out the foundation for future projects, emphasis on a well-thought-out design is therefore important. As motivated in the previous section, it should support the extension with new interfaces like real-time industrial ethernet (e.g. EtherCAT) or custom Aurora based communication links like OPAL-RT Opal Remote IO Network (ORION) [6]. The interface to RTDS is just one of many possible extensions and is presented here as an example.

[1] Previous publications also named it Simulator-to-Simulator Server, Sim2SimServer or S2SS.

Models on the FPGA should be implemented with XSG or HLS to provide unexperienced FPGA users a simple design entry method. An integration with the existing *VILLASnode* software simplifies the usage by combining both the *VILLASnode* and *VILLASfpga* configuration in a single file. At the same time, all the existing interfaces which are supported by *VILLASnode* can be connected to the new FPGA-based interconnect. *VILLASnode* provides an easy-to-use C Application Programming Interface (API) for controlling and data exchange with the FPGA framework. To leverage the performance of a FPGA implementation in conjunction with the flexibility of *VILLASnode* a bridge between the FPGA and the Central Processing Unit (CPU) is required. PCIe is the only user accessible, low-latency, high-bandwidth interface in modern x86-based machines. It has to be employed for synchronization and data exchange between the FPGA and CPU.

There is a number of industry standards protocols for Supervisory Control and Data Acquisition (SCADA) and substation automation tasks. The most prominent one is IEC 61850 which specifies Generic Object Oriented Substation Events (IEC 61850-8-1) (GOOSE) and Sampled Values (IEC 61850-9-2) (SV). These standards often define data structures which are far to complex for the task of real-time co-simulation. Most power system simulators have built-in support for those protocols to test existing automation equipment and understand the influence of the protocols in the system[2]. Due to the lack of alternatives, previous digital co-simulations often used those protocols. However, none of them is actually designed for this purpose and none of them has the notion of a simulation time step. There is no standardized protocol which supports the synchronization with the accuracy of a simulation time step in the range of 10 to 50 µs. In real-time co-simulation (not co-simulation of communication protocols), the interface and the underlying protocols are not the subject of the experiment. In fact, they are just the tool to enable them. Ideally, their presence ideally would not have an impact onto the results of the simulation. However, due to inevitable communication latencies this is not the case. Special interface algorithms and decoupling methods must be employed.

In some applications like thermal / electric co-simulation, the communication latency can be neglected as the thermal subsystem is running at much lower rate as the electric subsystem. In this thesis, we focus on the implementation of a single-rate interface as it has the most strict requirements. By demonstrating that these tight timing requirements can be met, we implicitly show that the same interface can also be used for less stringent demands.

1.2.1 Applications

In addition to the original goal, this framework envisioned to be used for the following use cases:

Real-time co-operative simulation The model is separated into two subsystems of which both are simulated in EMT-domain. One subsystem is simulated in

[2] E.g. for OPAL-RT: http://www.opal-rt.com/communication-protocols.

the real-time simulator while the other is simulated in real time by the Linux machine. The interface delay is compensated based on a Bergeron traveling-wave model of a transmission line.

Hybrid simulation The model is separated into two subsystems of which one is simulated in DP-domain. This enables simulation of large-scale subsystems in the phasor domain on the Linux machine or a dedicated High-performance Computing (HPC) cluster. Regions of interest can still be simulated in high resolution EMT-domain on the real-time simulator with a much smaller time step than the phasor subsystem.

Geographically-distributed simulation Co-simulation over large geographical distances will be affected by significant communication latencies. These latencies have to be predicted and compensated by using interface algorithms that can be based on Discrete Fourier Transform (DFT) and wavelet transforms. Especially on public networks like the internet these latencies are hard to predict. A central hub per laboratory can act as a gateway which implements those interface algorithms and manages data exchange.

User interaction & Monitoring Interaction with real-time simulations is usually done via vendor-specific tools like RTDS's RTDS Simulator Software (RSCAD) or OPAL-RT's RT-LAB. Most of them already provide APIs to interact with the simulator. Previous work has shown a web-interface for *VILLASnode* which allows users to monitor and control the simulation using standard web-browsers.

Big data analytics The presented framework offers the ability to record and replay signals continuously for each and every time step of the simulation in a customized and flexible manner. The collected results can be stored in Time-series Database (TSDB) for subsequent offline analysis. Big data analytic methods could be applied to study the vast amount of data. Existing record and replay features only offer a limited sampling rate and a limited number of signals.

1.3 Related work

During the work on this thesis, ACS and OPAL-RT demonstrated the first fully-digital interface between a RTDS rack and an OPAL-RT eMegaSim simulator [16]. For the first time, a synchronized and fully-digital co-simulation with only a single time step latency between both real-time simulators has been possible.

The OPAL-RTDS interface as well as the interface which is presented in this thesis are based on the *GTFPGA* block which has been block specifically developed in a collaboration between RTDS Technologies and Florida State University's Center for Advanced Power Systems (CAPS) in 2009 [22] [24]. Since then, it was used regularly used by CAPS for various research projects. Berger used it to implement a RTDS-to-Aurora bridge for Controller-Hardware-in-the-Loop (CHIL) testing [5]. Stanovich et al. implemented a Multi-agent testbed using a RTDS-to-Ethernet gateway [25]. Rentachintala presents a co-simulation interface between RTDS and OPAL-RT using a 11 bit parallel bus [21].

A combination of the GTFPGA block with PCIe interfaces was first mentioned by Stanovich et al. in [26] and by Hu et al. in [10] in 2013. Unfortunately, both publications lack details about their PCIe implementation.

RTDS supports distributed simulation across multiple RTDS racks by using a custom Inter Rack Communication switch (IRC) switch which is described in chapter 3.5. However their internal communication protocol is undocumented. The only possibility to get access to their simulator backplane is by using the GTFPGA block.

OPAL-RT itself offers Aurora-based communication interfaces using SFP fiber optics for their simulators[3]. Aurora (8B10B) is a data-link layer protocol developed by Xilinx for high speed communication. The primary use-case of the interface is for FPGA-to-FPGA communication or Modular Multi-level Converters (MMCs) applications. When carefully designed it could also be used to connect multiple RT-LAB nodes. However, the designated method for multi-target co-simulation is the RT-LAB software [17]. RT-LAB supports FireWire, Infiniband, Shared Memory (PCIe Scalable Coherent Interface (SCI)) as underlying communication protocols.

In 2006, OPAL-RT introduced Orchestra, a software communications framework for real-time data exchange between RT-LAB Simulink-based models and external software or hardware components [7]. Orchestra is built on top of the RT-LAB software and therefore uses the same communication channels as RT-LAB. A custom Orchestra API allows the integration of various software-based simulation tools like Matlab/Simulink, Dymola or MATRIXx. Orchestra is real-time capable and was inspired by the publish / subscribe concept of High Level Architecture (HLA) and uses Extensible Markup Language (XML) for Distributed Data Structures (DDS). A first application of this framework was presented in [18]. Paquin et al. present a real-time co-simulation of an All-Electric Ship (AES) which is performed by a total of three RT-LAB targets. Communication is done via Remote Memory Access (RMA) over a Dolphin SCI interconnect.

Finally, Flajslik et al. present techniques to reduce the PCIe communication latency for low-latency applications [9]. Using a FPGA prototyping board, they present PIO and polling techniques which enables a direct data-transfer between the FPGA and CPU without accessing the main memory of the system.

1.4 Structure

This introduction is followed by a brief introduction into the basics of real-time simulation. The theory of co-operative simulation and the required synchronization and communications schemes are covered in detail. Basic topics of modern computer

[3] http://www.opal-rt.com/aurora

architectures like interrupts and DMA are introduced as this work builds a bridge between between computer and power system engineering.

The next chapter 3 covers the architecture of the co-simulation framework. It starts off with the concept of the *VILLASnode* and continues by showing how this concept has influenced the architecture of the FPGA-based hub. The architecture describes main building blocks and their relationship. Different methods for data transfer and synchronization between the FPGA and the host CPU are compared.

The implementation chapter 4 contains detailed descriptions of the aforementioned components. It shows the design flow used for implementation and depicts ways to extend the framework with new components or models.

Functionality and performance of the framework is evaluated in chapter 5. Counter-based models are presented to calculate time step-accurate round-trip time measurements. Simple electrical models are used to showcase real-time co-simulations and hybrids simulations between the Linux host and RTDS.

Finally the last chapter 6 summarizes the result, shows some example applications and gives an outlook for possible future work.

2 Theoretical background

2.1 Real-time Simulation

A simulation is a representation of the operation or features of a system through the use or operation of another [27]. This thesis focuses on the digital real-time simulation of electric power systems. Models are executed on specialized real-time capable simulators or optimized computer systems. The most common approach for the simulation of EMT-based models are fixed time step solvers. They approximate the state of a time-continous model at discrete points in time whereas the state is solely depending on the results of previous calculations and current inputs. The period between two consecutive computations is called *time step*.

Real-time simulation refers to a model of a physical system that is executed at the same rate as the actual *wall clock* time. A fixed time step solver has the advantage that it can be executed in real time as long as the computation time does not exceed the time step. Case (a) of figure 2.1 shows a fixed time step simulation which meets this criterion. In case (b) the deadline for time step n is missed out which causes an overrun and the next step to be skipped. In comparison, the last case (c) displays an offline simulation. Time steps are executed immediately one after the other without idle delay in between.

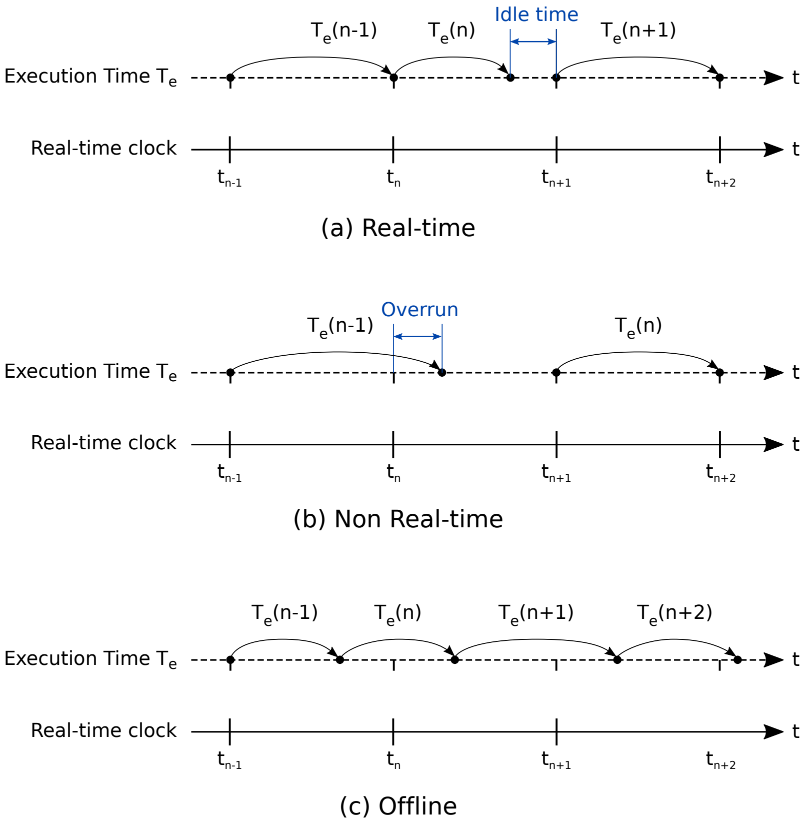
Both Faruque [8] and Belanger [3] summarize the current state of the art in real-time simulation electric power systems.

2.2 Co-operative Simulation

Co-operative simulation, or short co-simulation, describes the distributed simulation of multiple subsystems which form a coupled system. Every subsystem is modelled independently without detailed knowledge about details of the remaining system. Usually, systems are decoupled across spatial boundaries like a transmission line and only exchange a limited number of interface quantities. By limiting the amount of exchanged information, every subsystem can be solved with a dedicated method and / or timestep. In the context of power system simulation, a nodal admittance matrix has be decomposed into several smaller problems.

There are several motives that make co-simulation indispensable and helpful that at the same time may come with challenges:

- Different simulation tools often have to be integrated into a single simulation environment. Legacy models are sometimes hard to port to a new simulation



(a) Real-time

(b) Non Real-time

(c) Offline

Figure 2.1: Fixed time step simulation [8].

tool or certain models require specific libraries which are only available in a specific tool.

- Hybrid simulations may consist of EMT and DP-based subsystems. This requires decoupling and conversion between the time and time-frequency domain at subsystem boundary.
- Multi-physics simulations are an example where multiple time steps are used throughout the model as well as diverse simulation models. Thermal models are executed with time steps which are magnitudes larger than for electrical models. Multi-rate setups must be synchronized properly at the interface.

2.3 Soft & Hard Real-time Computing

Real-time computing describes hardware or software systems which are subject to a real-time constraint. This constraint is expressed in form of a deadline to which the system must complete certain computations or react to an event. For soft real-time systems, missing a deadline is acceptable but will cause in degradation of system performance. Examples are multimedia applications like video decoders or telephony. Hard real-time systems in contrast have to be designed in a way that deadlines are always met. Overruns are unacceptable and can only be caused by errors during system design. This restricts an engineer in the selection of algorithms. The amount of computation time for a hard real-time capable algorithms must be known in advance. This can be hard or even impossible to determine for iterative solvers which are therefore rarely encountered in real-time simulation.

2.4 Synchronization

Synchronization is the coordination of events in a distributed system in such a way that they coincidence in time. For most setups, synchronization information is signalled over the same link as data, and therefore equally affected by communication latency. This latency must be precisely estimated and compensated to guarantee a temporal coincidence of the synchronized events in both simulators. Network time protocols like Network Time Protocol (NTP), Precision Time Protocol (IEEE 1588) (PTP) or Inter Range Instrumentation Group Timecode B (IRIG-B) solve this problem by implementing algorithms like Christian's or Best Master Clock (BMC) [2].

The most commonly synchronized event between simulators are:

1. The start or end of a simulation case.
2. The beginning of a new time step.
3. The update or sampling of hardware Input / Outputs (IOs).

4. The completion of data exchange with other simulators.

A coordinated simulation start, ensures that results of the simulation are deterministic between runs and realistically represent a real world scenario. Only a synchronized start allows a transient response to be analysed. In AC systems, voltage and current sources generate their output in relation to a global reference phase per simulator. To couple certain active networks, the reference phases of involved simulators have to by aligned. This is usually guaranteed by starting both simulations at the same time.

The synchronization of time steps is necessary to avoid a shift between of simulator clocks. Without that, the time steps would drift apart after some time and the communication phases of both simulators would not be aligned. Furthermore, it would result in imprecise simulation results as the time step which is expected and used for the calculations does not match the real elapsed time.

The main motivation for real-time simulation is the ability to control and test real hardware. HIL experiments which are interfaced to multiple simulators require that all simulators sample their inputs at the same point in time. Sampling and the update of outputs is usually done periodically with the time step. As a result, those events are occurring relative to the time step event. In case no real hardware is attached to the simulators, the simulation actually can be simulated offline. In an offline simulation only the logical order of time steps must be observed.

In a local environment, simple optical Pulse per Second (PPS) signals or Digital IO (DIO) pins of the master simulator can be used to distribute the synchronization information. These signals are only affected by a very small latency which is usually far below 1 µs and jitter free.

However, if simulators are connected via an unreliable link like the internet, the communication latency is also affected by jitter which makes it hard to reliably estimate the latency. Under such circumstances, common time references like atomic clocks (Temps Atomique International (TAI)) or satellite navigation systems like Global Positioning System (GPS) or Globalnaja Nawigazionnaja Sputnikowaja Sistema (GLONASS) can be used as a synchronization source. They provide a highly accurate and stable clock as well as a reference to absolute global time.

2.5 Clock Hierarchy

Apart from simulation-related clocks, like time steps, there are typically other clocks involved in a distributed simulation as well. Processors, busses, memories, communication interfaces require their own dedicated clocking. The clock sources for those components are mostly fixed by the architecture of the simulator. As listed in table 2.1, only some of them are synchronized across the entire distributed system and others, like for simulator processors and interfaces, are not. This is due to the high clock speeds ($> 100\,\text{MHz}$) and architectural limitations.

Table 2.1: Clocks involved in heterogenous co-simulation.

Clock	Frequency	Synchronized?
Host CPU	3 GHz to 4 GHz	no
FPGA models	200 MHz	no
PCIe bus	125 MHz	no
AXI busses	100 MHz	no
Simulation time step	20 kHz to 100 kHz	yes
AC Power System	50 Hz and 60 Hz	yes

In a co-simulation, data needs to pass through several of those clocks. Whenever two clocks are asynchronous, this boundary is referred to as a Clock domain crossing (CDC). CDCs require special attention by the design engineer as they can cause meta-stability as shown in figure 2.2. A signal which is synchronous to a clock will always have a stable value at the rising edge of its clock. However, in a CDC the signal is captured by a flip flop which is synchronous to a different clock. If the rising edge of this destination clock happens to coincidence with the transition of the data signal, the resulting state of the destination flip flop is undefined and therefore meta-stable.

This problem must be resolved by synchronization circuits like a n-stage multi flop synchronizer shown in figure 2.3. By chaining multiple capture flip-flops which are clocked by the destination clock the probability for a metastable signal can be reduced. The n-stage flop synchronizer adds n cycles latency to the signal. However, if compared to the the simulation time step, this is negligible.

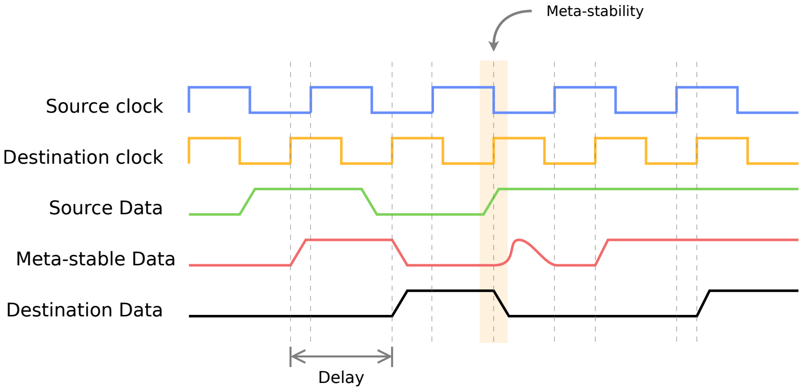


Figure 2.2: A meta-stable signal.

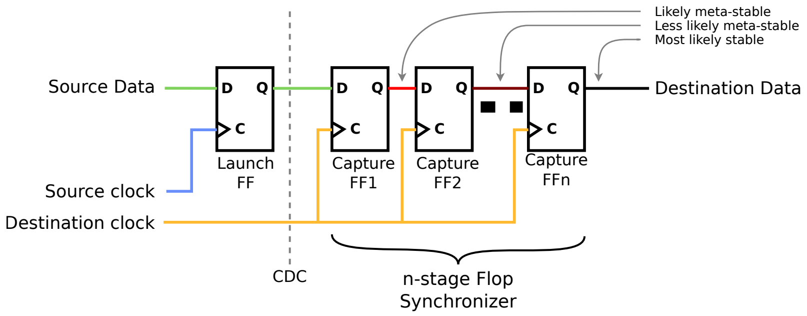


Figure 2.3: A n-stage Flop Synchronizer to mitigate meta-stability.

2.6 Scheduling

During a co-simulation, all involved parties periodically exchange interface quantities and compute their system solution for the next time step. The temporal order in which this is done can be altered for improving certain aspects of the interface.

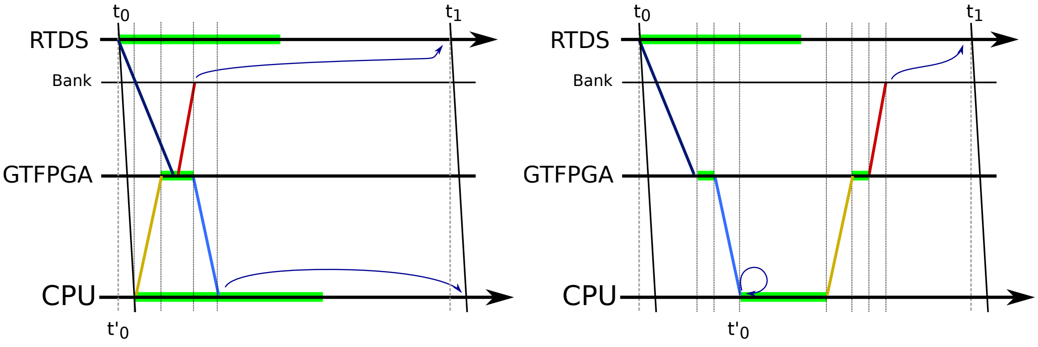
The simplest mode is shown in figure 2.4a, all simulators start their time step cycle with the exchange of interface quantities while at the same time starting to compute the solution for the next step. In this concurrent or parallel approach all exchanged quantities represent the state of the previous time step as the quantities of the current time step are yet to be calculated. This results in an inevitable communication latency of at least one time step between all simulators. This latency must be compensated by interface algorithms to guarantee system stability and simulation fidelity. For a single time step latency (typically 50 µs) transmission line models based on the travelling wave theory can be used.

Alternatively, a serial pattern can be used which is shown in figure 2.4b. Here the computation phase of two simulators has been serialized. This eliminates the single time step latency at the cost of half the available computation time per simulator and it is in contrast with main motivation behind distributed simulation concept. In contrast to the parallel exchange, the number of simulators which can participate in the serial pattern is limited by the amount of computation and latencies as they will sum up and may cause the deadline to be missed.

In a multi-rate simulation simulators run with different time step periods. The ratio between the rates still must be an integral value. A typical application is the co-simulation of a thermal - electrical systems where the thermal system runs with a much bigger time step. Figure 2.5 shows the communication pattern of a multi-rate co-simulation. Similar to the first two examples, serial and parallel patterns exists.

2.7 Computer Architecture

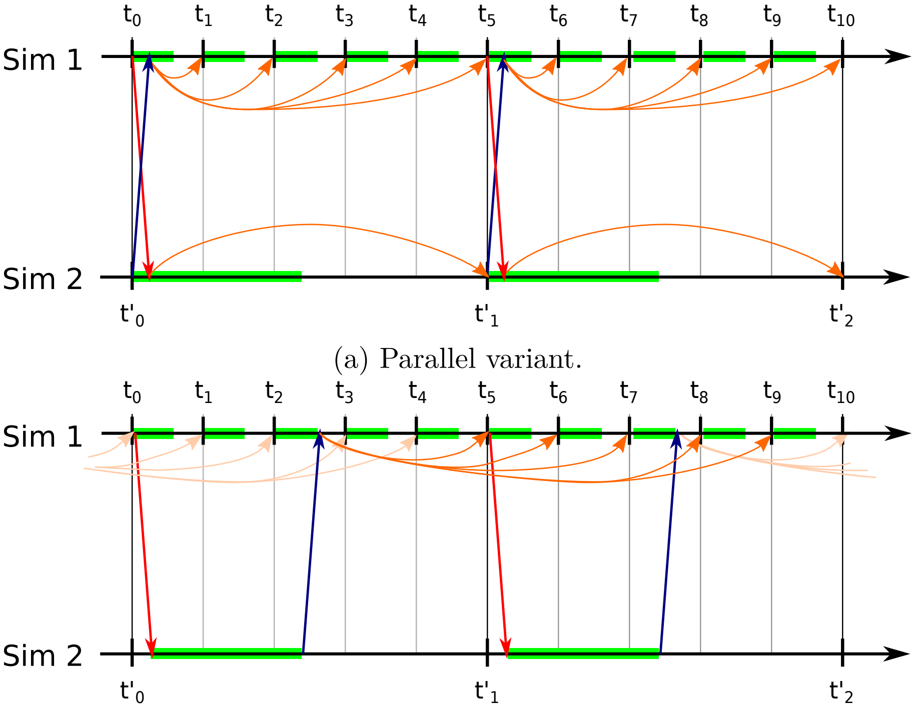
Modern computer systems are becoming more and more distributed. This applies for distributed clusters which can be found in HPC as well as for single worksta-



(a) Parallel variant.

(b) Serial variant.

Figure 2.4: Co-simulation communication patterns.



(a) Parallel variant.

(b) Serial variant.

Figure 2.5: Multi-rate communication patterns.

tions. These systems consist of multiple processors, a variety of peripherals like memory, network cards and disk controllers. Lately, co-processors like specialized Graphic Processing Units (GPUs), Intel's Xeon Phi or FPGA-based accelerators are supporting this trend.

When mapping simulations onto such architectures, especially real-time simulations, a closer look to synchronization and data transfer between the involved components is required.

2.7.1 Direct Memory Access (DMA) & Programmed IO (PIO)

With PIO and also called Memory-mapped IO (MMIO) the CPU issues single read and write transfers for each data word. This can be done by mapping the device memory into the physically addressable memory of the CPU. For PCIe devices this mapping is configured by changing the so called PCIe Base Address Registers (BARs). The CPU can only read and write word by word which limits has the disadvantage that the CPU is kept contiguously busy similarly to polling which is described in the next subsection.

As a consequence, DMA has been invented to avoid this. As of today, most peripherals have their own DMA controllers on-board. DMA is a technique which offloads the data transfer task to a dedicated controller. The CPU instructs the DMA controller to perform a data transfer by writing to its device registers using the previously described PIO. After initiating the transfer, the CPU can directly continue with other tasks until the completion of the transfer is signalled to the CPU by one of the two previously described mechanisms.

Simple DMA controllers only support a single outstanding transfer. Once a data transfer has been configured and started, the CPU must wait for completion before the subsequent transfer can be configured. More sophisticated DMA controllers support Scatter-Gather (SG) operations. They can increase the performance of many small transfers and facilitate transfers and configuration at the same time. SG extends the simple DMA controller by adding support for multiple outstanding operations. A dedicated memory region is shared by the CPU and DMA controller to keep a queue of outstanding and completed of transfers. This queue contains *buffer descriptors* which can be seen as little work packages. Every buffer descriptor contains a source and destination address and the length of the transfer in bytes.

2.7.2 Polling & Interrupts

Since the early age of microprocessors, interrupts are the standard method to synchronize peripheral devices with the main processor. Common examples are Network Interface Controllerss (NICs) which notify the CPU about new incoming data or input devices like mice and keyboards. Interrupts are triggered by asserting a special interrupt pin of the processor. This causes the CPU to pause its current task by branching into an interrupt sub routine. Because the interrupt line is shared between all peripherals the Interrupt Service Routine (ISR) has to determine which of those devices caused the interrupt. It then dispatches the interrupt service request to a device-specific handler which is part of the Operating System (OS) driver. The handler then has to react to the interrupt condition and initiate further actions.

Linux introduced a concept called *soft interrupts* which can postpone the execution of device-specific interrupt handlers to a later stage. The hardware ISR only acknowledges the interrupt and schedules a *tasklet* to handle the data intensive tasks. This mechanism has the advantage that it increases the number of interrupts which can be handled.

Often it is needless to interrupt the CPU for every event which occurred on a device. A good example are devices which generate a high amount of interrupts like network cards. The arrival of a network packet is usually worthy to trigger an interrupt. However, under high loads this would interrupt the CPU quite often and hence limit the system performance. A concept called *IRQ coalescing* reduces the load by delaying the interrupt until a certain amount of events occurred. The benefit of this technique heavily depends on the requirements of the application. In real-time applications coalescing is usually undesired because it artificially increases the latency. Yet it can be used to increase the throughput of bandwidth intense network applications.

The alternative to interrupts is *polling*. In contrast to interrupts, it does not require special hardware like interrupt lines or controllers. Polling detects events by repeatedly reading status registers of the memory of the device. This periodic reading occupies the CPU and is therefore often avoided because no other tasks can be executed while waiting for a new event. However, it can lead to a reduction of interrupt latency because there is no overhead caused by the execution of interrupt handlers.

3 Architecture

This chapter describes key components and their relationship in the *VILLASnode* and *VILLASfpga* frameworks. It starts by introducing a concept which is common to both projects. The RTDS interface is of particular importance as it is one of the main objectives for this work. The architecture is an effort to design a future-oriented framework around this interface which can be extended with other interfaces or FPGA-based models. Figure 3.1 shows a very high abstraction of the architecture. It consists of the existing RTDS simulator and a new Xilinx VC707 FPGA board which is inserted into a Intel x86_64 computer.

3.1 Concept

The following concept abstracts complex co-simulation setups with can consist out of two or more subsystems. It hides slight differences in the architectures of *VILLASnode* and *VILLASfpga* from the user. This unified abstraction allows the integration of these frameworks. A single configuration file can be used to configure the complete setup.

Key components in a distributed simulation are subsystems and interfaces. In the following description the terminology *nodes* and *paths* is used respectively.

A *super-node* is an instance of one of the frameworks (*VILLASfpga* or *VILLASnode*). It controls and monitors several *nodes* and the *paths* across them. *Super-nodes* can be connected by special *node-types* which are called data-movers. An example for such a data-mover is a User Datagram Protocol (UDP) / IP based socket connection for the connection of two *VILLASnode* instances over a network

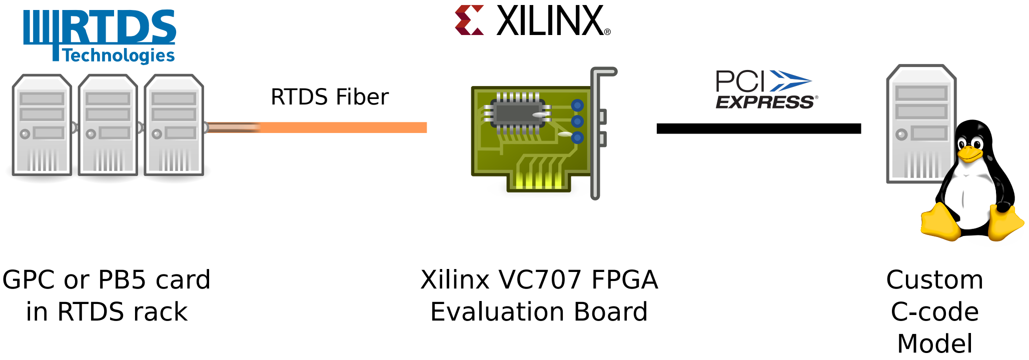


Figure 3.1: High-level Architecture of RTDS to Linux interface.

or a DMA controller for the data transfer between *VILLASfpga* and the host system which runs *VILLASnode*.

A *path* is a unidirectional pipe between *nodes*. Every path must have a single source node and can have multiple destinations. The path periodically reads samples from the source nodes and sends them to all destination nodes. Synchronization is done implicitly: A source node will block the read operation as long as there is no new data available or the processing is still underway.

A *node* can be a physical simulator, a HIL experiment, sensors like Phasor Measurement Units (PMUs) or virtual models. Every node is an instantiation of a specific node-type and exists in a *super-node*. Most nodes have a single input and output at which they can receive and send simulation data. Usually a node should not artificially delay the processing in addition to the time they require for solving or transmission (in case of a datamover). Hence, multiple nodes can be chained together while still being able to process a sample of simulation data in a single time step. However, there is one exception to this rule: Node-types which are interfacing real hardware must align their processing to a time step boundary. This is required to guarantee a synchronized update and sampling of real world signals across all involved nodes.

As shown in figure 3.2, this concept enables the user to build arbitrary complex topologies for co-simulation.

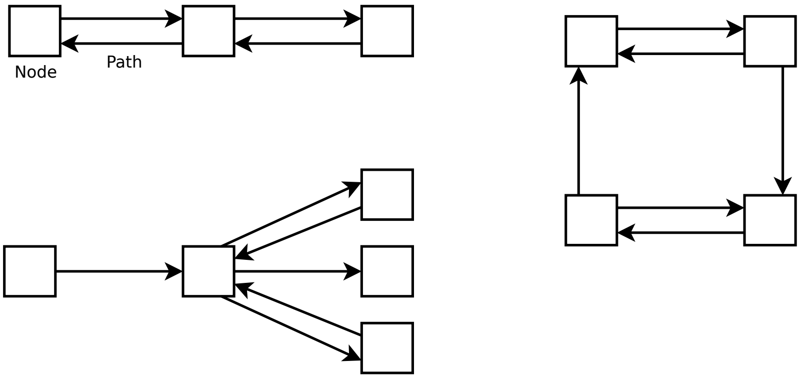


Figure 3.2: Possible co-simulation topologies which can be realized.

Figure 3.3 shows an example for a full-featured setup of *VILLASnode* and *VILLASfpga* consisting of three super-nodes and three simulators.

There are six main data paths in this example:

1. Simulator #1 ↔Simulator #2 via ①, ⑪ - ⑬
2. Simulator #1 ↔Simulator #3 via ① - ⑩
3. Simulator #1 →Time-series Database (TSDB) (Hadoop) via ① - ⑧, ⑯ - ⑰

4. Simulator #3 →Time-series Database (TSDB) (Hadoop) via ⑩, ⑨, ⑯, ⑰
5. Simulator #1 →User Interface (WebSockets) via ①, ②, ⑪, ⑭, ⑮
6. Simulator #3 →User Interface (WebSockets) via ⑩ - ⑤, ⑭, ⑮

Those paths are made up of several user-defined HLS / XSG and SW-based models: ②, ③, ⑤ ⑦ and ⑧. In this example, the internet link between super-nodes #1 and #2 ⑥ is coupled in phasor domain. Nodes ⑤ and ⑦ are models which handle the transformation between DP- and EMT-domain.

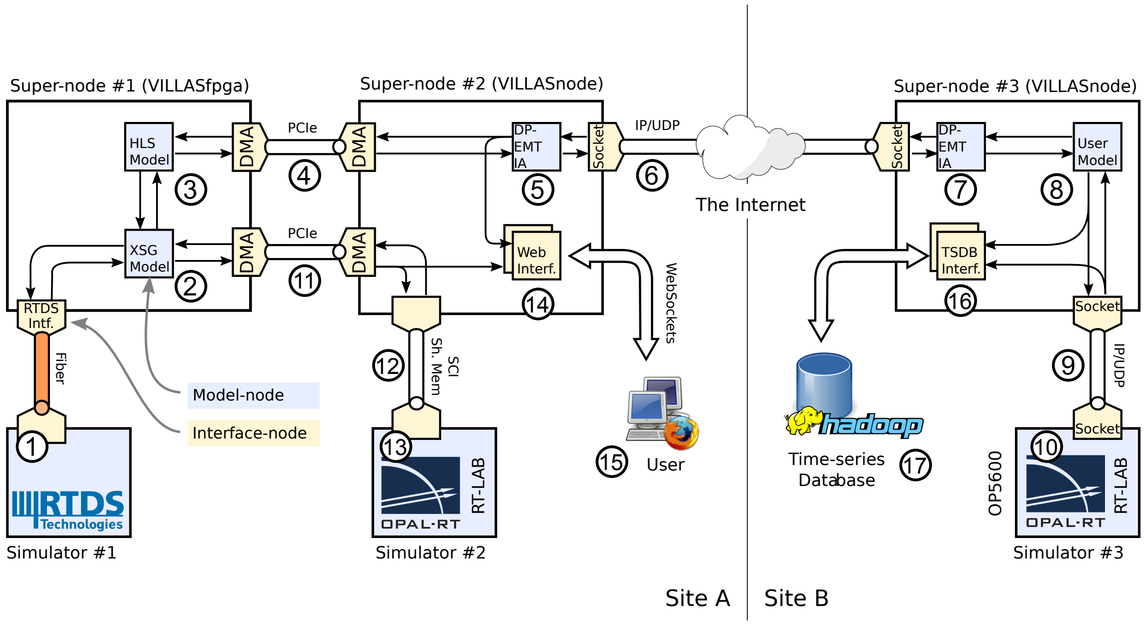


Figure 3.3: Example of integrated setup consisting of two *VILLASnode* super-nodes and a *VILLASfpga* instance.

3.2 VILLASfpga

In the *VILLASfpga* framework, nodes are realized by IP blocks which are instantiated in the user FPGA-design. The type of nodes which are available for the *VILLASfpga* differ from the ones which are available for *VILLASnode*. Figure 3.4 shows the high level architecture. In this thesis, node-types for HLS and XSG models and the RTDS interface have been implemented. The concept of a path are realized by a FPGA-internal switching network which is based on AXI4-Stream busses (yellow).

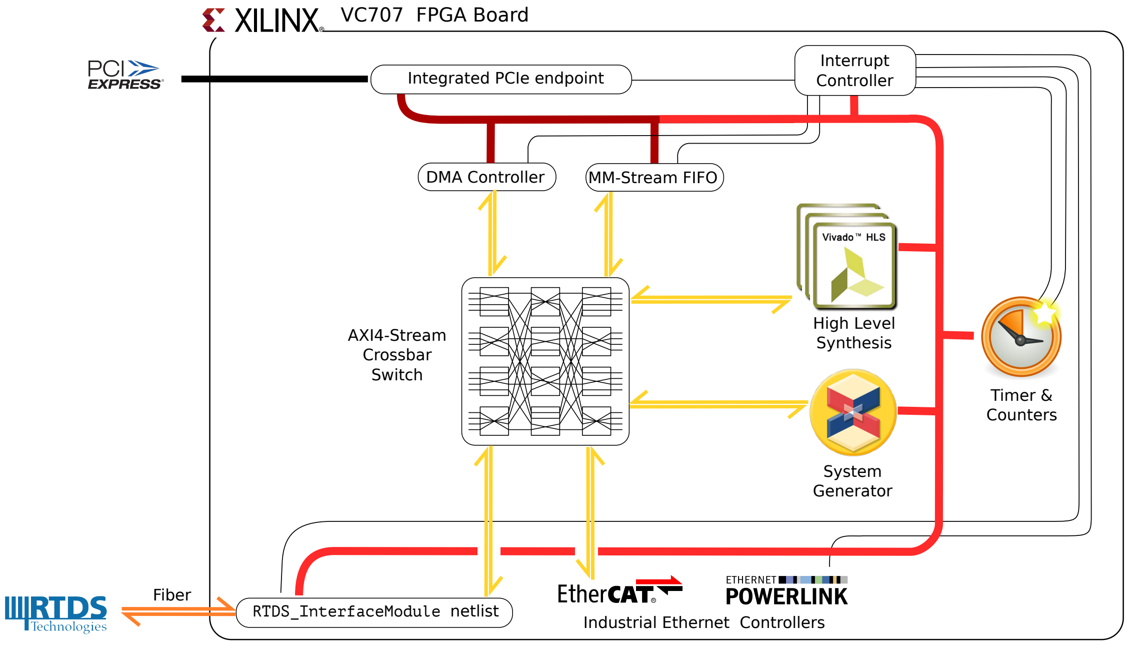


Figure 3.4: The *VILLASfpga* architecture.

3.2.1 FPGA Evaluation Board

VILLASfpga is based on a VC707 evaluation board from Xilinx. The board is interfaced via PCIe to a host machine which itself runs *VILLASnode* and used for monitoring and controlling the *VILLASfpga* framework. This selection was made based on the availability of boards which are supported by RTDS for its GTFPGA interface. RTDS is selling the VC707 board as part of their MMC Support Unit[1]. There are also versions of the interface which are compatible with older Virtex 5 and 6 FPGAs (ML507 and ML605 boards). However only the newest 7-series FPGA families are supported by Xilinx' new Vivado development environment which will be extensively used due to its support for AXI4-based interconnect networks. Furthermore, the old FPGAs are only supported by the older Xilinx Integrated Synthesis Environment (ISE) development environment which is discontinued.

3.2.2 Interconnect

Figure 3.4 shows several internal and external interconnect networks used by the FPGA framework.

PCIe (black) is used for interfacing the FPGA design to the host system. It allows direct memory access from CPU to FPGA and vice versa. PCIe is a memory addressable bus. Every read and write transaction is targeting a specific address or address range. In the FPGA, the PCIe endpoint is implemented by a

[1] https://www.rtds.com/the-simulator/our-hardware/mmc-support-unit/

hard-macro which is implemented in dedicated silicon instead of using generic FPGA resources. On the host system, the PCIe root-complex is part of the CPU or the IO Controller Hub (ICH).

AXI4 (dark red) is a memory-mapped bus as well. It is used only internally by the FPGA. All PCIe transactions are translated to AXI4 transactions. Only high bandwidth data-movers are connected to this bus.

AXI4-Lite (red) is a simplified version of the AXI4 interconnect without the support for burst transfers. It is primarily used for register access if only single-beat transfers are required. An example for such a register access is the configuration of an IP block like the DMA controller or the change of parameters of a *node*.

AXI4-Stream (yellow) is used for streaming packets between the *nodes* in the *VILLASfpga* framework. Unlike the memory-mapped busses, transactions are not targeting an address. They consist of one or many data-beats which are grouped to packets.

The Advanced eXtensible Interface Bus (AXI) busses are standardized by the Advanced Microcontroller Bus Architecture (AMBA) specification [1]. AMBA is a standard introduced by ARM in 1996 which defines a family of busses: Advanced High-performance Bus (AHB), Advanced Peripheral Bus (APB), AXI, AXI Coherency Extensions (ACE). Most ARM-based systems nowadays are premised on the latest generation of AXI4 interconnect.

Because of its wide-spread adoption as an industry standard, AXI4 busses have been chosen for the implementation. AXI4 busses meet all the requirements for the transfer of small packets in this application. In most use-cases they reach 100 % of the theoretical peak bandwidth.

Other protocols like IBM's Processor Local Bus (PLB) or OpenCore's Wishbone[2] bus have been considered as well. The decision for AMBA busses has been made due to the good availability of AXI-compatible IP by Xilinx's new Vivado Integrated Development Environment (IDE). Xilinx provides AXI infrastructure IP like interconnects, DMA controllers, clock or data width conversion free of charge.

Some of Xilinx's earlier FPGA families included hard-macro processors based on the PowerPC architecture. PowerPC is an architecture by International Business Machines Corporation (IBM) and uses the PLB as its main interconnect. Beginning with the 7-series family, Xilinx made a transition to ARM processors for their newer SoCs (Zynq). This change was accompanied with the migration from PLB to AMBA busses and the introduction of the Vivado IDE. Vivado is the successor of Xilinx's ISE and Xilinx Embedded Development Kit (EDK).

Other bus openly specified bus systems like the Wishbone bus have not been an option due to their limited support and compatibility with the Vivado IDE.

[2] http://opencores.org/opencores,wishbone

AXI and PCIe are not buses in the traditional sense as they do not use a shared medium between all connected devices. Modern interconnection networks, also called Network-on-Chip (NoC), are only using individual point-to-point links between devices and interconnect switches. Transactions on these links are always initiated by a single master and targeting a single slave. This applies to both the memory mapped AXI4 bus, as well as the AXI4-Streaming bus. Multiple IP blocks can be linked by dedicated interconnect IP.

AXI4-Stream

Streaming busses have an important role in the *VILLASfpga* framework. Thus, they will be covered here in detail.

An AXI4-Stream interface consists of usually five signals:

axi_clk is the interface clock. All other signals are synchronous to this clock meaning that they are sampled at a rising edge of this clock.

axi_tvalid is asserted by the master in the same clock cycles as data signal `axi_tdata` holds a valid value.

axi_tready is asserted by the slave as soon as it is ready to accept data. A data beat is transferred as soon as both the `axi_tvalid` and `axi_tready` are simultaneously asserted. `axi_tready` might depend on `axi_tvalid` but not the other way around.

axi_tlast is asserted by the master together with `axi_tvalid` to mark the end of a packet.

axi_tdata is driven by the master and contains the data beats. Xilinx requires this signal to have a width which is a multiple of 8 bit. In *VILLASfpga*ăall data signals are 32 bit IEEE-754 single precision floating point numbers.

Figure 3.5 shows a transfer of a single packet on an AXI4-Stream link. AXI interfaces are using a two-way handshake between the master and the slave. A transaction is initiated by the master by driving the first data word to `axi_tdata` and asserting `axi_tvalid`. The transfer of this first beat is completed by the slave as soon as it asserts `axi_tready`. This allows the slave to throttle the data-transfer by de-asserting `axi_tready` even during the transmission of a packet. If there is no backpressure from the slave, AXI streaming links can transfer a single data beat per clock cycle. *VILLASfpga* is using a data width of 32 bit for the `axi_tdata` signal and a bus `axi_clk` of 125 MHz. This results in a bandwidth of around 475 MiB s^{-1} for the node-to-node links.

The big advantage of the streaming links is their low latency and minimal resource usage. This is a requisite for Audio / Video (AV), network and simulation applications like *VILLASfpga*. In constrast to the memory-mapped AXI4 busses there are no additional clock cycles required for the addressing.

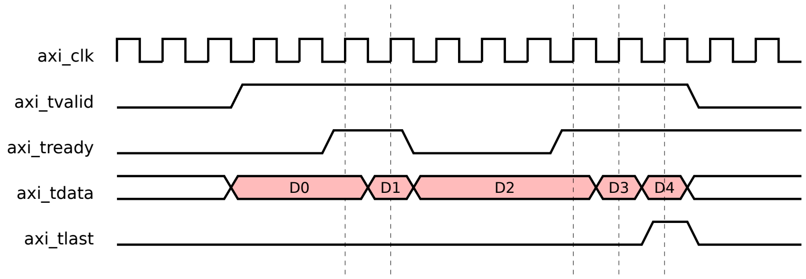


Figure 3.5: An AXI4-Stream transaction with four data beats.

In the scope of the *VILLASfpga* framework, every time step a sample is transferred as a single AXI4-Stream packet. Multiple signals in a sample are transferred as individual data beats of this packet.

The connection of multiple *nodes* can be realized with a crossbar switch. The usage of a crossbar switch is optional but has the big advantage that routing between all connected nodes can be reconfigured during run-time. This gives the user the ability to test multiple different variants of a certain model or quickly switch between multiple configurations without resynthesizing the FPGA bitstream. Figure 3.6 shows 16 *nodes* which are connected to a crossbar. In the example, the crossbar is configured to route packets from source node C over hops P and O to the destination F. Figure 3.6a shows a logical abstraction of the same configuration. Xilinx's AXI4-Stream interconnect IP core supports up to 16 master and slave interfaces.

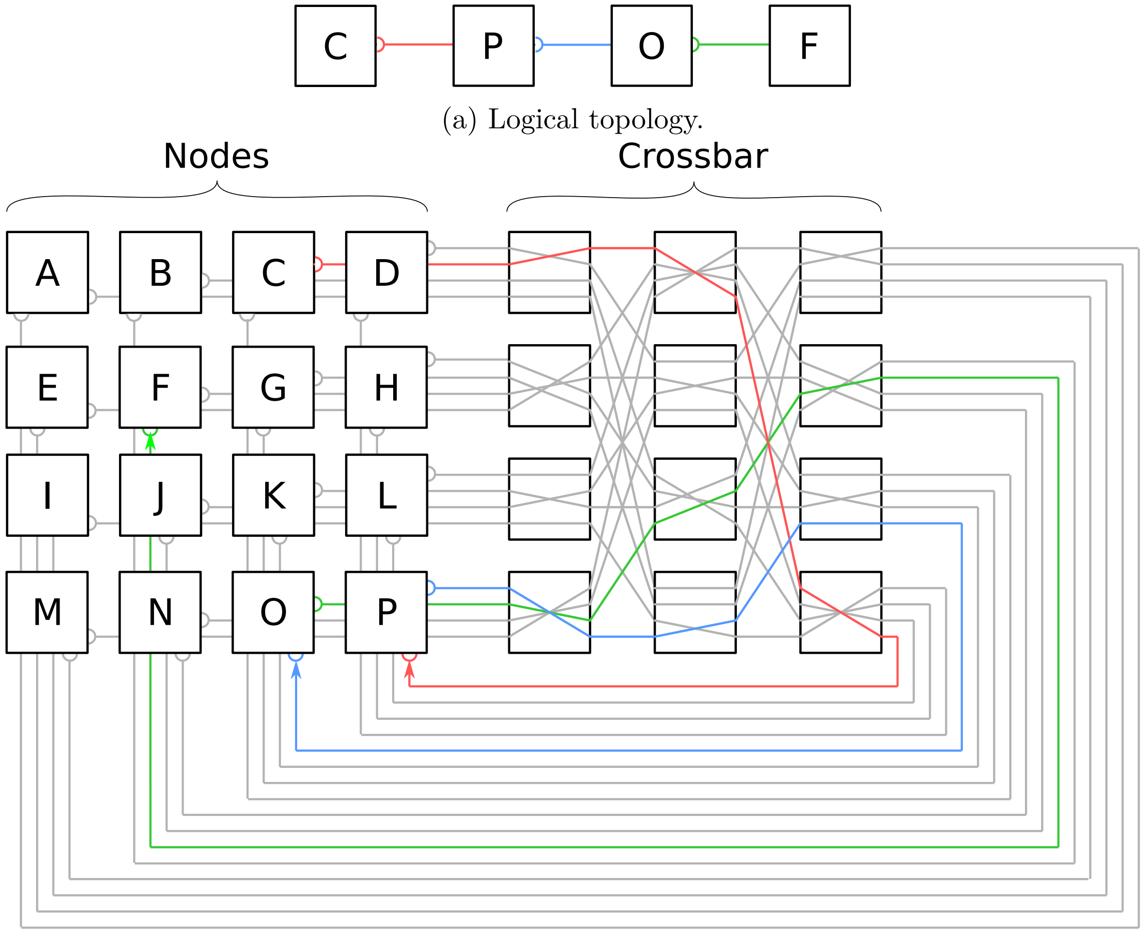
3.2.3 Data-movers

As stated earlier, *VILLASfpga* is using PCIe as its interface to the host machine. Because this is a memory mapped bus, special node-types, the data-movers, are required which translate the memory-mapped data on the AXI and PCIe busses to packets on the AXI4-Stream links. Chapter 2.7.1 introduced two different approaches for data transfer between the FPGA and the CPU. To compare their performance, three different data-movers have been tested: Both a simple and a SG DMA controller as well as a First-in First-out (FIFO) queue which is controller via PIO. Every datamover is realized as a node-type as part of the VILLAS concept.

VILLASfpga is not limited to a single instance of those data-movers. If desired, the user has the choice which and how many data-movers will be instantiated in the design. However, for most use cases a single data mover is sufficient.

3.2.4 Models

FPGA-based models profit from clock cycles as low as 5 ns and nearly unlimited parallelism offered by the nature of Programmable Logic (PL) devices. Often used in Rapid Control Prototyping (RCP) and simulation of power electronic converters, these models use very small time step periods. For such applications, a model which



(a) Logical topology.

Figure 3.6: Connectivity options of a 16-by-16 AXI4-Stream switch crossbar.

has been previously simulated in *VILLASnode* or another simulator can be moved to the FPGA. In this case, the PCIe interface would no longer be used for data exchange but only to monitor and control the FPGA models.

To exploit the full capabilities of FPGAs, custom Hardware Description Language (HDL) code like VHSIC Hardware Description Language (VHDL) or Verilog has to be written. However, implementing bigger models using HDL is cumbersome and error prone. Instead, more user-friendly and graphical RCP tools like Mathwork's Simulink, or National Instrument's LabVIEW are used. This ongoing trend in digital design raises the level of abstraction and is often referred to as Electronic System-level Design (ESL) design. It hides details of the Register Transfer Level (RTL) from the designer and enables them to focus on the design and behaviour of their model. Yet, the designer has still to keep in mind the underlying FPGA architecture to obtain an efficient implementation. A critical decision is the selection of a suitable number format to discretize continuous quantities. Xilinx is providing their own optimized Simulink blockset called Vivado System Generator for DSP (XSG) which assists the designer in those considerations.

Alternatively, Mathworks offers a product called Simulink HDL coder. It can transform any Simulink model consisting of standard blocks into HDL code. This simplifies the adoption existing models to a FPGA implementation. Though, the results of Simulink HDL coder are not optimized for a specific FPGA architecture. In fact, the generated HDL code is generic and could be implemented on most FPGAs or even in hard silicon.

Vivado High-level Synthesis (HLS) is another method introduced by ESL design. It uses high-level programming languages as design entry. With certain restrictions, HLS tools will transform this high-level language into HDL code. Xilinx supports both C and C++ languages in its HLS tool. HLS is a very young technology. Xilinx released the first version of HLS in 2013. Major improvements can be seen with every released version of the tools.

In the course of the *VILLASfpga* implementation, both simple HLS and XSG models have been implemented to provide a starting point for more complex models designs in the future.

3.3 VILLASnode

VILLASnode implements the previously described concept as a software application which runs on a Linux machine. Figure 3.7 shows several of the currently supported node-types. Until now, *VILLASnode* has been mainly used as a gateway. However, simple models can be implemented as a custom node-type using C code. In addition *VILLASnode* is also used to control the *VILLASfpga* PCIe card which are plugged into the same system. In this case, every datamover between *VILLASfpga* and *VILLASnode* appears as a node in both frameworks. For the best performance *VILLASnode* has been written in the C programming language and optimized for

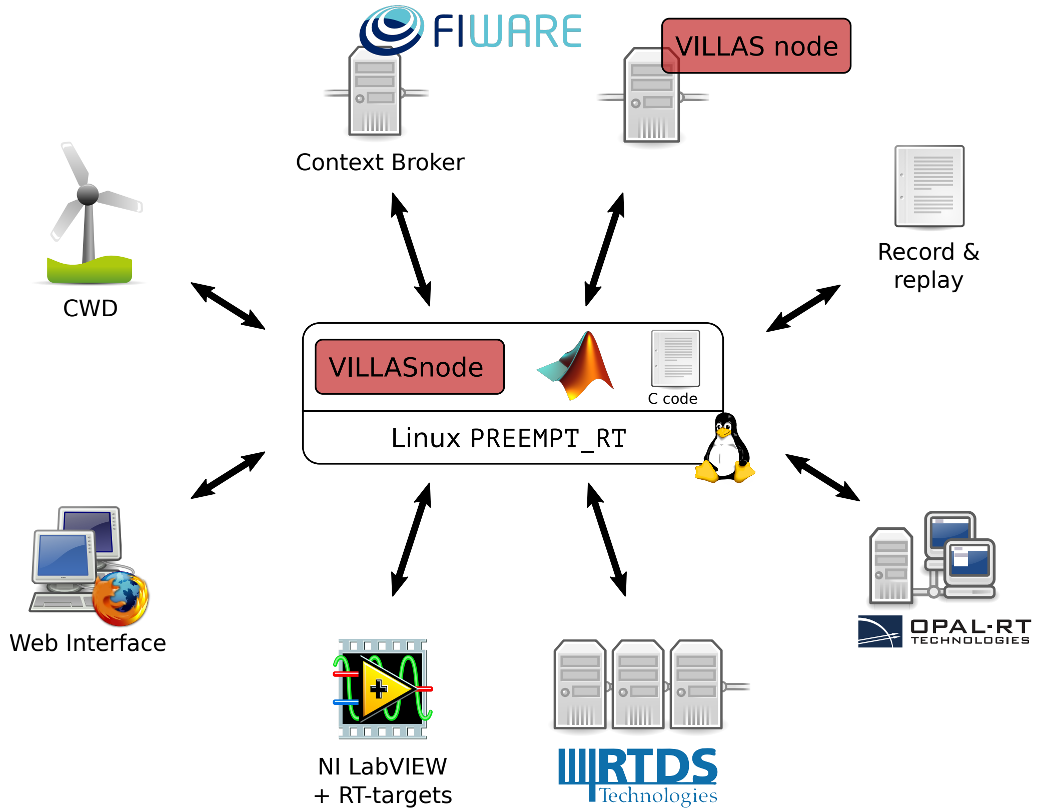


Figure 3.7: Overview of supported node-types by *VILLASnode*.

real-time operation. So for example, every path is processed in a dedicated thread to guarantee a non-blocking operation.

As a little addition to the previously described concept, *VILLASnode* supports hook functions which can be assigned to paths. These hook functions are executed for every sample of simulation data. They can be used to alter the signals, e.g. applying a simple filter operation, or filter the data based on various criteria. Most of the node-types for example can not handle simulation time steps as low as 50 µs. Instead a hook function is employed to reduce the interface rate. Other use cases of hook functions are the collection of statistics, monitoring of the link qualities the implementation of interface algorithms.

Figure 3.8 shows an example setup for internet-distributed simulation. Here every site runs a dedicated *VILLASnode* instance. Locally available simulators are connected to their corresponding *VILLASnode* instances. Communication between the sites is handled by *VILLASnode* which is acting as a gateway.

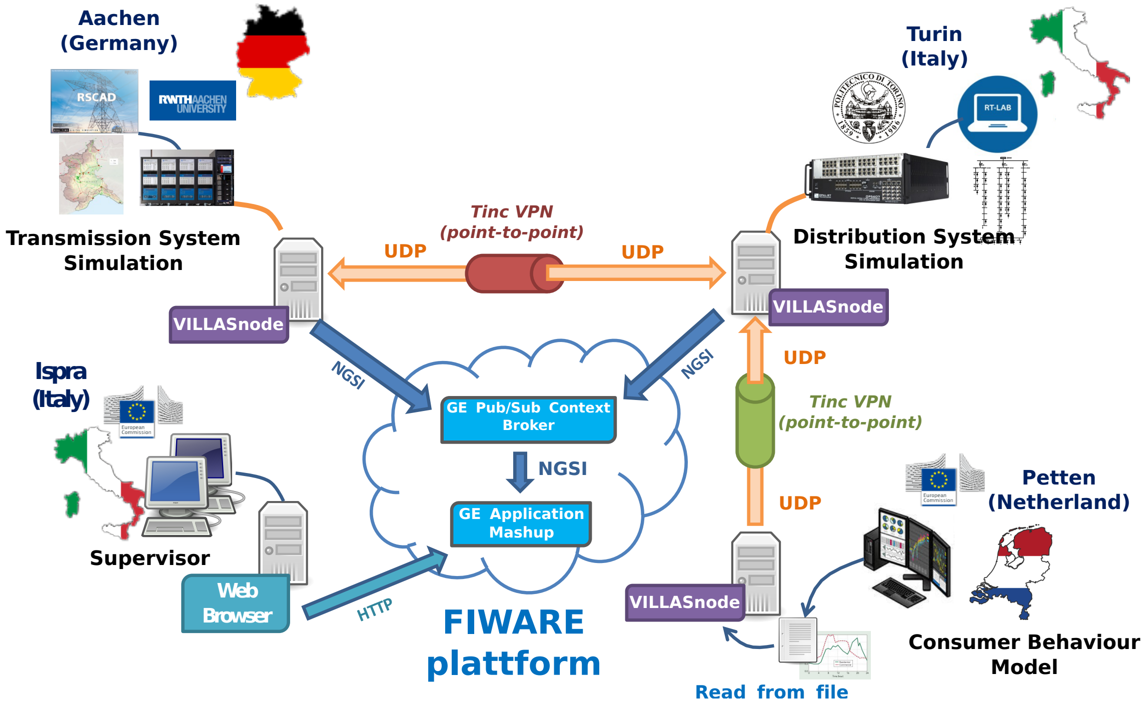


Figure 3.8: European Real-time Integrated Co-simulation laboratory (ERIC-LAB) demonstration.

3.4 Host Machine

The host machine's main task is to run the *VILLASnode* software and to house the *VILLASfpga* PCIe-card. To have plenty of margin for models being executed in the *VILLASnode* framework, a recent Intel x86_64 multi-core system was chosen. A multi-core machine allows the concurrent execution of multiple paths / models on different cores. The only alternative with approximate performance are ARM-based systems. However, they rarely feature full-sized PCIe slots which are required for the VC707 FPGA-board.

Instead of a dedicated FPGA-board, an integrated FPGA-CPU System-on-Chip (SoC) like Xilinx's Zynq architecture could be selected. It combines a PL with two ARM processor cores (PS). Both parts of the architecture are tightly interconnected by a total of nine AXI busses. It would provide the best connection between *VILLASnode* which would run on the ARM cores and *VILLASfpga* which would be implemented in the PL. Unfortunately, this FPGA architecture is not officially supported by RTDS for its GTFPGA interface. Even though there are Zynq-based SoC's which are using the same Virtex-7 architecture for their programmable logic as the VC707 board, it is questionable whether the RTDS interface would work. Most parts of the *VILLASfpga* architecture are designed with the adaptability to Zynq SoC's in mind. For deployments where power consumption or size are critical, both *VILLASnode* and *VILLASfpga* could be ported to a Zynq-based system without larger changes.

The host machine runs a Fedora Linux OS for the execution of *VILLASnode*. Linux is not a real-time OS by default. Hence, careful optimizations and tuning are indispensable. The most important change is a `PREEMPT_RT`-patched kernel. `PREEMPT_RT`[3] is patch-set maintained by Thomas Gleixner to improve the real-time performance of the vanilla[4] Linux kernel.

The *VILLASnode* code and its portion which controls *VILLASfpga* is running in the Linux userspace. Device drivers are usually part of the OS and therefore run in kernelspace. However, this would cause a big number context switches between the userspace application and the kernelspace driver. To eliminate this overhead the driver was implemented as part of the userspace application. Yet, the driver still needs to request resources from the OS: FPGA device memory needs to be mapped into the userspace application and interrupts have to be forwarded. Linux is providing two APIs which enable the implementation of PCIe device drivers in userspace. By using one of those standard APIs an unmodified Linux kernel can be used. This simplifies the maintenance and setup of the host machine. Both the UIO and VFIO APIs have been considered and are presented in the following sections.

3.4.1 Userspace IO (UIO)

UIO is Linux' first API for device drivers in userspace. It was introduced to Linux in 2007 with version 2.6.23[5].

The UIO framework allows that certain parts of the driver can still remain in kernel while most of it is implemented in userspace. The kernel stub part of the UIO driver exposes its capabilities via `ioctl()` and `mmap()` system calls to the userspace.

For most Peripheral Component Interconnect (PCI) devices the generic `uio_pci_generic` kernel module can be used. It exposes access to the PCI BAR via `mmap()` and allows the passthrough of legacy PCI interrupts.

A major limitation of UIO is the fact that PCI devices are not allowed to act as a bus master. Bus mastering describes the ability of a PCI device to initiate transactions on the bus. Acting as a master allows the device to access host memory or other PCI devices directly. This is considered dangerous if not controlled by the operating system itself and therefore disabled by UIO.

Without the ability to access host memory, DMA transfers are impossible. MSI interrupts rely on bus mastering as well and are therefore disabled. Without PCIe Message Signalled Interrupt (MSI) only the legacy interrupt mechanism is available which does not support vectors. To multiplex different interrupt sources an additional interrupt controller is then required.

[3] https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html

[4] A vanilla kernel is the unmodified upstream version of the Linux kernel.

[5] http://git.kernel.org/git/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=beafc54c4e2fba24e1ca45cdb7f79d9aa83e3db1

3.4.2 Virtual Function IO (VFIO)

VFIO is a newer API for accessing devices directly from userspace. It was introduced to Linux in 2012 with version 3.6 [6].

It has been primarily designed to improve the performance of PCI devices in Kernel-based Virtual Machine (KVM). KVM is a full virtualization solution for Linux which runs Virtual Machines (VMs) as a userspace processes. VFIO provides a secure way to assign a PCIe device to a VM. To enhance the performance of those PCIe devices they must be able to utilize their DMA controllers which in turn requires that the PCIe devices must have bus mastering capabilities. Though, as stated earlier, bus mastering can be dangerous as the PCIe device gains full control over the system. In case the PCIe device is assigned to a VM or an userspace application, an attacker who controls either one of those can gain access to the host system. With the introduction of Intel's VT-d virtualization technology, Input / Output MMUs (IOMMUs) and PCIe root complexes gained the flexibility to effectively isolate PCIe devices from each other and the host memory. By isolation, the device can be limited to access only certain parts of the address-space which belong to the controlling userspace process. Thus the PCIe device is prohibited to cause harm by accesses to the host kernel memory or other VMs.

Furthermore, VFIO provides a comprehensive `ioctl()`-based API to create DMA memory mappings, register MSI interrupts and more. DMA mappings and BAR accesses are created by using the `mmap()` system call. While UIO always requires root permissions, VFIO can make use of Linux permissions system to allow arbitrary users access to the PCIe device. This could be advantageous in environments like HPC clusters where root access is restricted.

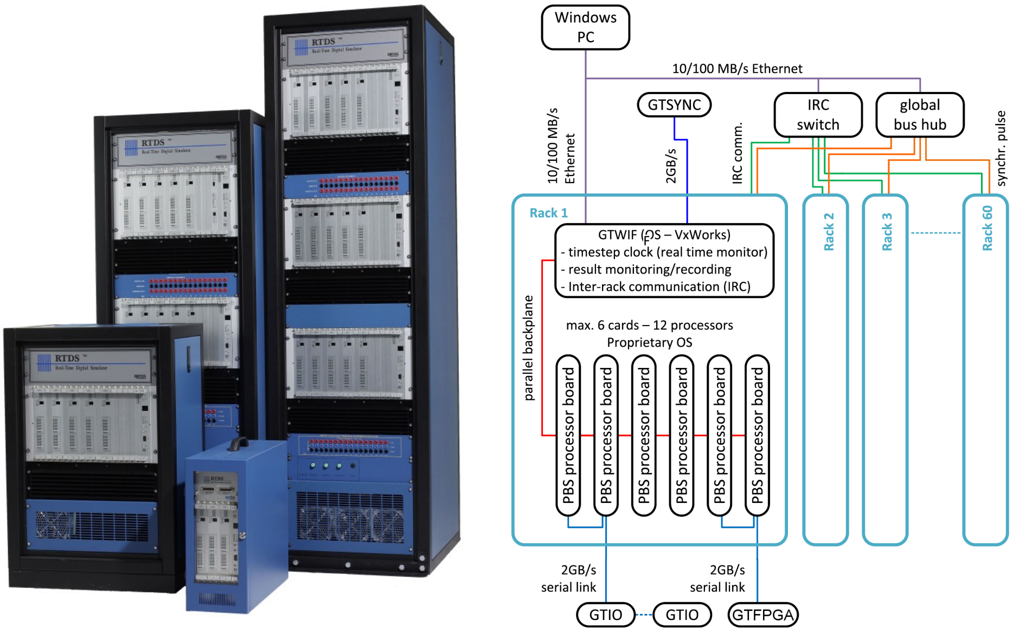
3.5 Real-time Digital Simulator (RTDS)

To demonstrate the functionality and performance of *VILLASfpga*, an interface to the RTDS has been implemented. This interface can be seen as an example for the extensibility of the *VILLASfpga* framework. As it is described below, RTDS is based on custom hardware which also holds for the simulation software environment. As such, it is a perfect example of mitigating interface issues of a vendor specific solution based on a generic framework proposed in this thesis.

This sections starts by presenting the architecture of the Real-time Digital Simulator (RTDS). RTDS is one of the two big players for digital real-time power system simulation. With its introduction in 1993 RTDS was one of the first commercial fully DRTS on the market [14].

Figure 3.9a shows a typical setup of a RTDS simulator which can be built up of one or more cubicles. Each cubicle houses one or more racks. Each rack can be used

[6] https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=cba3345cc494ad286ca8823f44b2c16cae496679



(a) Cubicles configurations.

(b) Architecture [8].

Figure 3.9: The Real-time Digital Simulator.

standalone or in combination with others for the simulation of larger systems. In case of a multi-rack simulation an IRC is used to exchange simulation data and a Global Bus Hub (GBH) is used to synchronize the time step between racks.

Each rack can contain up to six processing cards which are interconnected by a backplane. Processing cards are the core of each simulator. They compute control signals and the power system solution. RTDS is therefore continually updating its processing cards. Table 3.1 gives an overview of existing generations of RTDS processor cards.

A Giga Tranceiver Workstation Interface card (GTWIF) controls the time step of a rack and coordinates communication over the rack backplane. It also handles inter-rack communication and controls the rack by an Ethernet connection from a RSCAD workstation.

Figure 3.9b shows the architecture of RTDS.

Table 3.1: Generations of RTDS processor cards.

Year	Abbr.	Name	#[7]	Processor	Speed
1993	TPC	Tandem Processor card	2	Analog Devices ADSP21062	12 MHz
1997	3PC	Tripple Processor card	3	Analog Devices ADSP21062	40 MHz
2002	RPC	RISC Processor card	2	IBM PowerPC 750CXe	600 MHz
2005	GPC	Giga Processor card	2	IBM PowerPC 750GX	1 GHz
2011	PB5	PB5 Processor card	2	Freescale MC7448	1.7 GHz

Recent processing cards support the attachment of peripherals via *Giga Tranceiver (GT)* fiber optic connections. Commonly found extensions modules are Giga Tranceiver Input Output card (GTIO) or GTNET. A variety of different input / output cards (GTDI, GTDO, GTAI, GTAO, GTFPI) enable the attachment of HIL hardware to the simulator. The GTNET card allows data-exchange via standard protocols like IEC 61850 Sampled Values, GOOSE, UDP, Transmission Control Protocol (TCP) and more.

The following subsections highlight extension cards which are important for the interface between RTDS and the co-simulation framework.

3.5.1 Giga Tranceiver Network card (GTNET)

The GTNET extension card provides a real time communication link to and from the simulator via Ethernet. Several industry standard protocols like IEC 61850-9-2 SV, GOOSE, Distributed Network Protocol (DNP3) or custom UDP / TCP protocols are supported via exchangeable firmwares. The latest version of the GTNETx2 card consists of two GTNET modules. Every module can execute a certain firmware which enables simultaneous operation of two different protocols. Other features include record and playback of large datasets from the workstation machine. The card offers 100/1000 Copper, 100BASE-FX, or 1000BASE-SX fiber connections for the interface to Ethernet networks.

At first glance this makes the GTNET card a perfect candidate for interfacing RTDS to other simulators. Unfortunately, it is impossible to synchronize the interfaces to the internal time step which is a premise for a tight coupling of two simulators. The next section introduces an alternatives interface which supports synchronization.

3.5.2 Giga Tranceiver Field Programmable Gate Array (GTFPGA)

The Giga Tranceiver Field Programmable Gate Array (GTFPGA) board allows users to interface their custom FPGA designs via fiber optics to GPC or PB5 processing cards. Every time step, the user design can exchange up to 64 signals per

Table 3.2: GTNET protocols with their supported number of signals and sending rates.

Firmware	Format	Values	Rate
GTNET-SKT	UDP / TCP	300 60	1 kHz 5 kHz
GTNET-PMU	IEEE C37.118		
GTNET-GSE	GOOSE	4 TX / RX modules à 64 pts.	
GTNET-SV	SV	4 streams (à 4 V + 4 I) 1 stream (à 4 V + 4 I)	80 pts. / period 256 pts. / period
GTNET-DNP3	DNP3	1024 binary output pts. 512 binary input pts. 500 analog output pts. 100 analog input pts.	1 kHz 1 kHz 10 Hz 10 Hz
GTNET-PLAYBACK	COMTRADE	8 channels	20 kHz

direction with a RTDS processing card. The start of a simulation time step is signalled in-band over the fiber optic. This facilitates the time step synchronization of FPGA designs with models executed on the simulator which is unique to the GTFPGA interface.

The GTFPGA interface consists of several components:

1. Several blocks which can be integrated into the RSCAD model. They provide an interface control signals which then can be connected to controller voltage and current sources. The primary use-case of the GTFPGA interface is controlling MMCs. Several blocks for the small times-step Voltage Source Converter (VSC) mode exist.

2. A closed-source netlist which can be integrated into a user FPGA design. This netlist is currently only available for Virtex 5, 6 and 7 FPGAs.

3. A Xilinx FPGA evaluation board. RTDS is currently supporting the ML507, ML605 and VC707 boards which are based on the above mentioned FPGAs. Figure 3.11 shows the new VC707 FPGA board which is sold by RTDS as part of their MMC Support Unit[8].

The interface is implemented by incorporating the synthesized [9] netlist in the user-defined FPGA design. By shipping a compiled netlist, RTDS protects its proprietary protocol which they use on the Giga Tranceiver (GT) fiber connections. RTDS is providing reference design for all of the supported boards as a starting point.

[8] https://www.rtds.com/the-simulator/our-hardware/mmc-support-unit/

[9] The process of compiling a VHDL description of a design into a gate-level description.

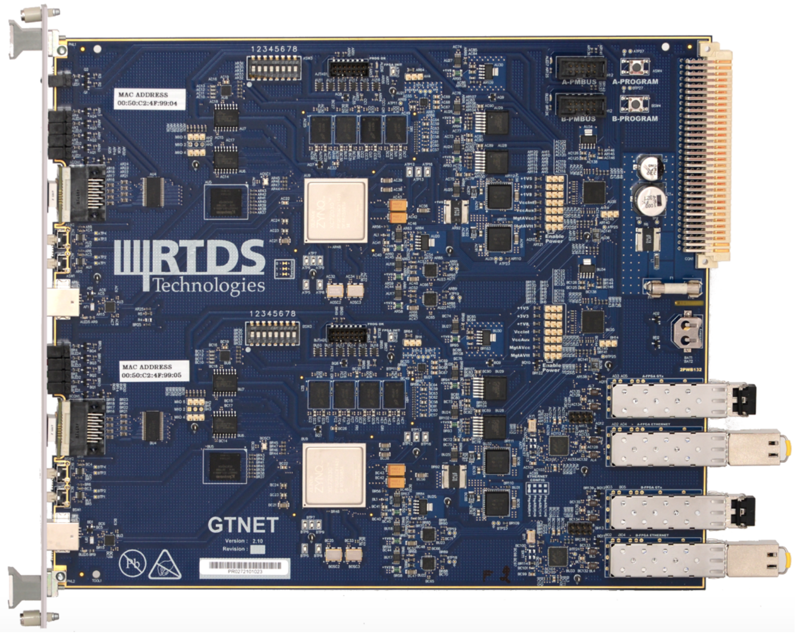


Figure 3.10: The GTNETx2 extension card for RTDS.

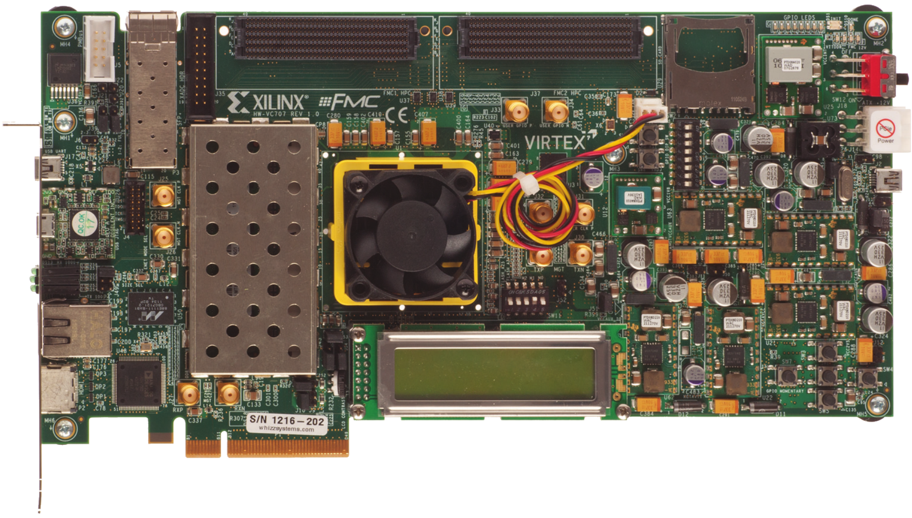


Figure 3.11: The Xilinx VC707 evaluation board.

3.5.3 Giga Tranceiver Synchronization card (GTSYNC)

The GTSYNC extension card allows the synchronization of the simulation time step to an external clock [23]. Unlike the other extension cards it is connected to the GTWIF which is controlling the rack time step.

Figure 3.12 displays the GTSYNC card with multiple inputs and outputs for synchronization signals. The card supports IRIG-B, PTP and PPS sources and can relay the synchronization to several optical and coaxial PPS outputs.

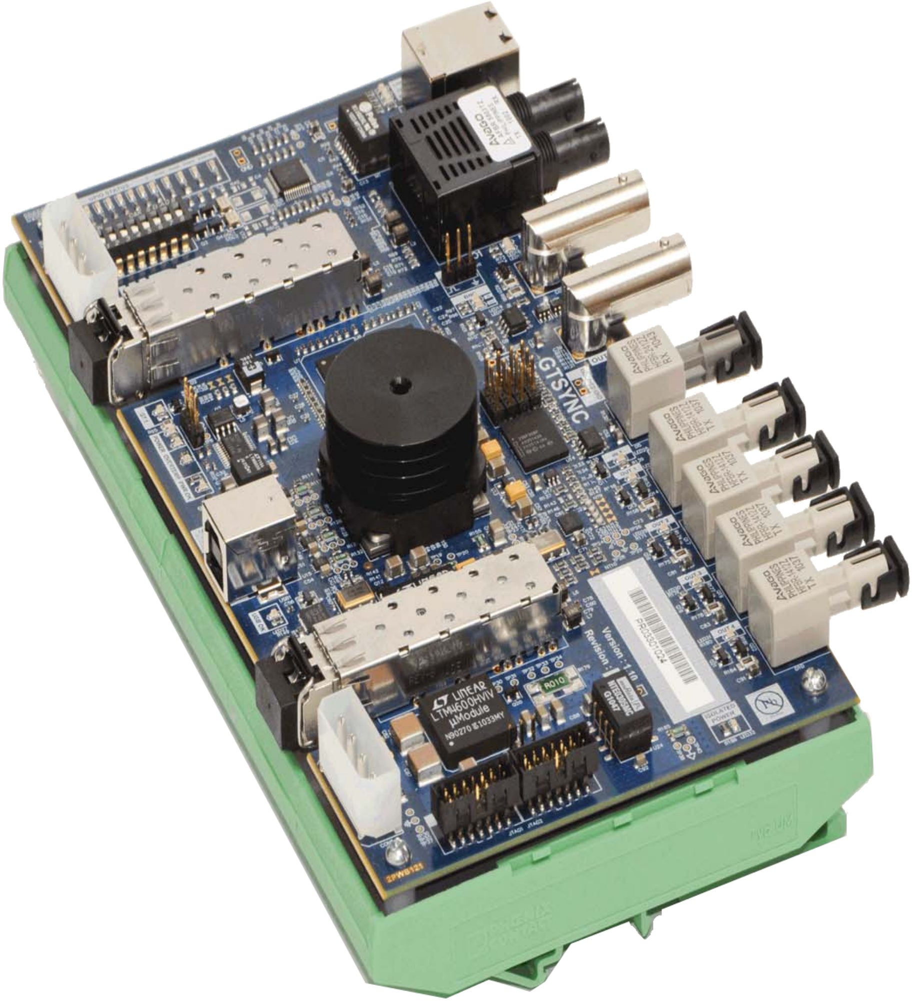


Figure 3.12: The Giga Tranceiver Synchronization card (GTSYNC).

Main use-cases of this card are:

1. Reducing the clock drift of the internal GTWIF oscillator which is used to generate the time steps.
2. Synchronization to a global time reference which is required for the test of PMUs.

For co-simulation it is necessary to control the time step of all participating simulators. In first experiments, RTDS was distributing its time step as a master to all connected devices. This can be done by using a digital IO pin or the in-band synchronization signal from the fiber optic GTFPGA interface. For more complex setups, it might be necessary to synchronize RTDS as a slave by providing the master clock from the *VILLASfpga* board. According to RTDS, their simulator can be externally synchronized by using the 1PPS or IRIG-B inputs of their GTSYNC card. As long as the time step frequency is a round number (20000 Hz / 50 µs or 16000 Hz / 62.5 µs), the time step which occurs at the 1-second-mark will align with the synchronization pulse of the PPS input. However, the start of the simulation time step can be affected by jitter of up to 1 µs.

4 Implementation

VILLASfpga's architecture shares several building blocks which can also be found in standard Network Interface Controllerss (NICs). This chapter is focused on implementation details of those and presents other components in the *VILLASfpga* framework. A major part of this work is focused on the communication between the FPGA board and the CPU via PCIe which will described in another section. The last part of this chapter presents a design flow which has to be used for adding new models or interfaces to the framework.

4.1 FPGA Evaluation Board

The Xilinx VC707 FPGA evaluation board features a wide range of connectivity options. Within this thesis, only the PCIe and SFP interfaces are used. Both interfaces require very high link speeds with up to 5 Gbit s^{-1}. The Virtex FPGAs provide specialized IO transceivers for such high speed interfaces. Other components including flash and Random-Access Memory (RAM) memories, LEDs, a Liquid-crystal Display (LCD), push buttons, DisplayPort and Ethernet connections remain unused.

Figure 4.1 shows the card inserted in one of the PCIe slots of the host machine. The board is build around a Virtex-7 XC7VX485T-2FFG1761C FPGA [39]. This specific model supports up to 56 Gigabit Tranceiver (GTX) of which 27 are accessible on the VC707 board:

- 8 wired to PCIe edge connector
- 1 wired to the SFP+ cage
- 1 wired to Serial Gigabit Media-independent Interface (SGMII) connection to the Ethernet PHY
- 1 wired to Sub-Miniature-A (SMA) connectors
- 2x 8 wired to FPGA Mezzanine Card (FMC) HPC connectors.

Other features of the FPGA include [30]:

- 485760 Logic cells which are grouped into 75900 Configurable Logic Blocks (CLB).
- 2800 DSP slices for hard-macro arithmetic operations

- 37080 kBit of on-chip memory
- 4 hard-macro PCIe blocks
- 56 GTX
- 700 User IO pins
- 1 XADC analog to digital converter block

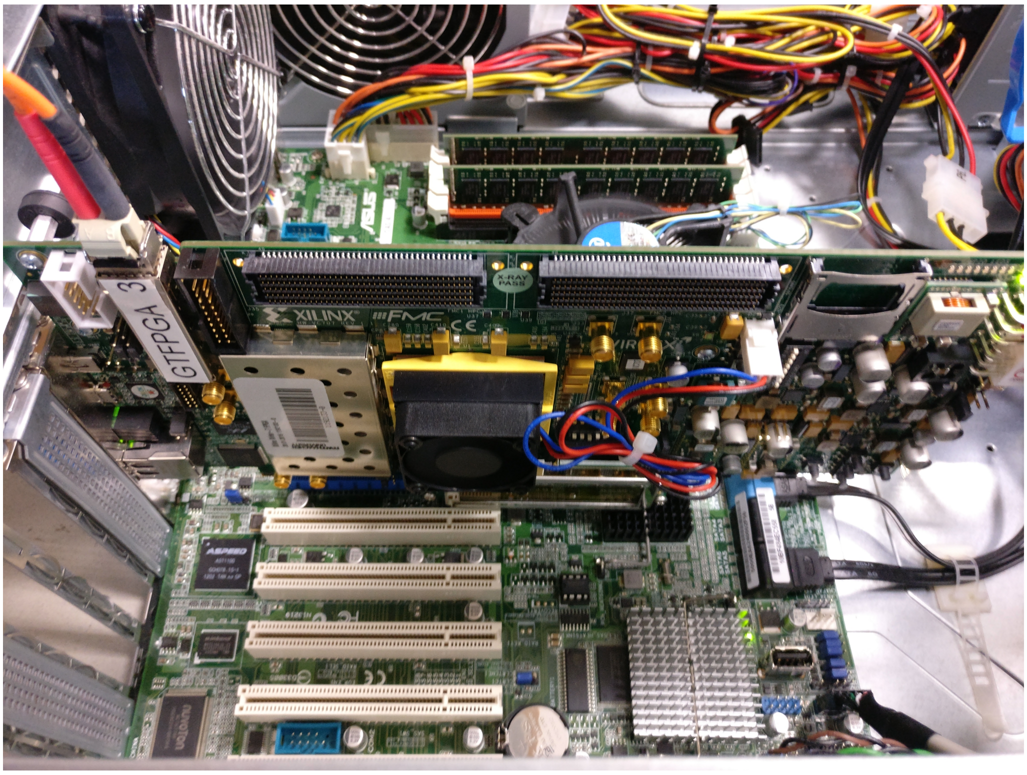


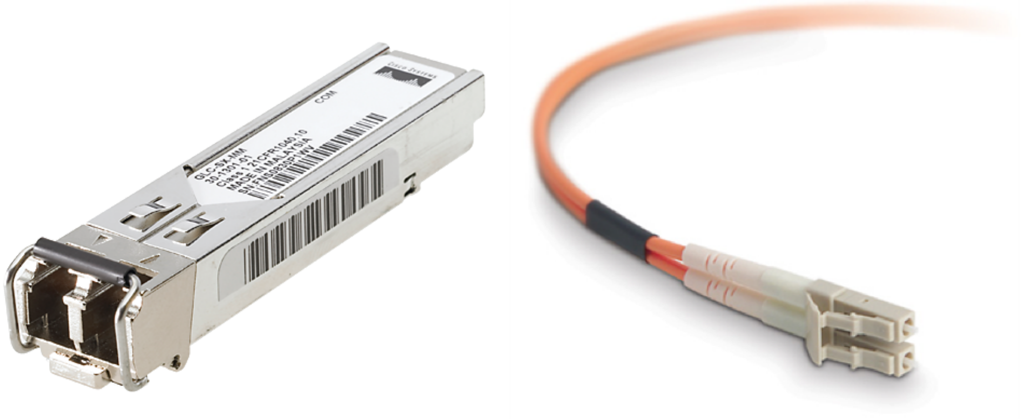
Figure 4.1: The VC707 board plugged into a PCIe slot of host machine.

4.1.1 SFP Fiber Module

Small Form-factor Pluggable (SFP) modules are an industry standard for exchangeable network interface modules. It specifies both an electrical and mechanical interface for those modules. Most common SFP modules like the one shown in figure 4.2a are house optical and electrical tranceivers for 1000BASE Ethernet which can be found in most modern network switches, routers or interface cards.

Within this thesis, SFP modules are used to connect the FPGA board to the RTDS simulator. Inter alia, SFP modules are used by RTDS for the connection between processing and extension cards. Their modules are using a link speed of

2.5 Gbit s^{-1} and standard Lucent Connector (LC) fiber connections like depicted in picture 4.2b.



(a) A SFP optical tranceiver module.

(b) A multi-mode fiber with Lucent Connector.

Figure 4.2: RTDS connectivity technology based on LC connectors.

The VC707 board provides access to 16 additional GTX via two HPC FMC extension connectors. This enables the extension of the board with up to 16 additional SFP cages with modules like shown in figure 4.3.

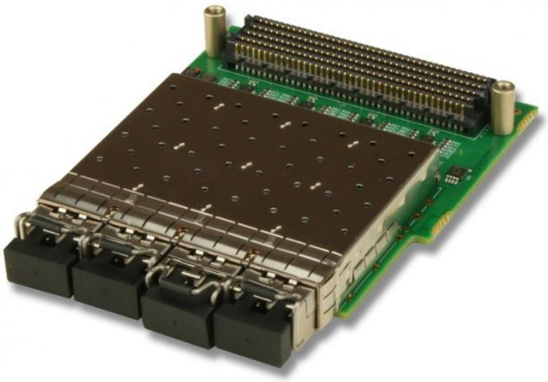


Figure 4.3: A FMC add-on module with four additional SFP cages.

$$
\begin{aligned}
\Delta t &= 10us && \text{Time step} \quad (4.1)\\
N &= 64 && \text{Values / Time step} \quad (4.2)\\
M &= 4Byte && \text{Bytes / Value} \quad (4.3)\\
B &= N * M/\Delta t = 25,6MiB/s && \text{Bandwidth} \quad (4.4)\\
U &= 500MiB/s/B = 5\% && \text{Utilization} \quad (4.5)\\
& && (4.6)
\end{aligned}
$$

Figure 4.4: Estimate bandwidth requirement of *VILLASfpga*.

4.1.2 PCI Express (PCIe) Interface

PCI Express (PCIe) is a high throughput and low-latency interface used between the FPGA board and the host system. The VC707 board supports up to eight second generation PCIe lanes with a link speed up to 5 Gbit s^{-1}. A bandwidth of 500 MiB s^{-1} can be achieved per lane. This results in a maximum throughput of 4 GiB s^{-1} which is far more than required by *VILLASfpga* as proven by equation 4.4. The *VILLASfpga* implementation is using 4 lanes.

PCIe is a serial point-to-point interface. Multiple PCIe Endpoints (EPs) like the FPGA board are connected a single PCIe Root Complex (RC). Every connection is made up of several lanes to maximize the throughput. Sometimes additional switches are used to extend the topology as shown in figure 4.5.

Like the SFP cage, every PCIe lane is driven by single GTX of the FPGA. The Virtex 7 FPGA features a integrated EP which is implemented as a hard macro. A hard-macro is an application-specific implementation in silicon. By using it usage of general FPGA logic ressources is reduced and performance is increased. It processes the serial data link layer and frames incoming data into PCIe Transaction Layer Packets (TLPs).

TLP packets are then translated by an AXI-to-PCI bridge. This bridge is an IP block provided by Xilinx and implemented in standard FPGA logic resources. It translates PCIe into AXI transactions and thereby allows memory accesses in both directions. AXI masters in the FPGA gain the ability to read and write to memory locations in the main system memory or other PCIe devices. In the same manner as the CPU or other PCIe device can access memory or registers of AXI slaves.

Both the AXI and PCIe bus support burst transfers to accelerate the transfer of continous memory ranges. The AXI-to-PCIe bridge does is performing well at translating those burst transfers between the two protocols. However, burst transfers on the PCIe bus are limited by page size boundaries. This means that they must not cross a 4 kB aligned address.

Data-movers which will be introduced in the next section must take care of that limitation as it only exists on the PCIe bus. Yet, the IP cores are designed for the AXI4 bus which does not impose this restriction.

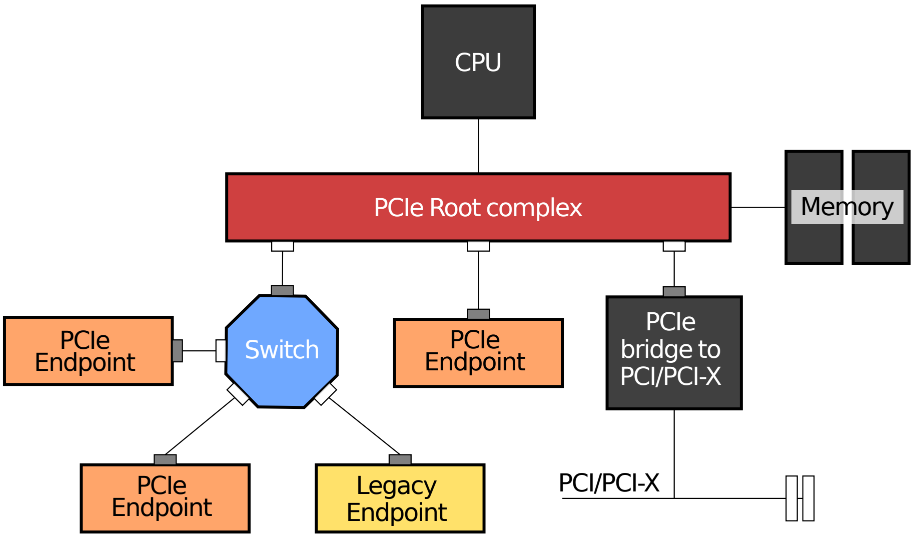


Figure 4.5: Example PCIe topology [19].

4.2 IP cores

The *VILLASfpga* design is composed of several IP building blocks or IP cores. Table 4.1 lists cores used by *VILLASfpga* which have been provided by Xilinx or have been self implemented.

The upper part of this table made up of node-types as described in section 3.1. These IP cores always feature at least a master and slave AXI4-Stream interface (AS) to exchange sample data with other nodes. Most of them also include an AXI4-Slave interface (AM) for adjusting parameters via registers. Some node-types require synchronization with other components of the system via interrupt lines (I).

4.2.1 Data-movers

Data-movers are special node-types which facilitate the data transfer between the *VILLASfpga* and the *VILLASnode* domains. Hence, they can be also seen as an interface or pipe with two ends. One end appears as a node in the *VILLASnode* configuration, the other end exists in the *VILLASfpga*.

The host system is using addressable memory to store and exchange simulation data. Whereas the FPGA is using AXI Stream links to connect the nodes. This requires the data-movers to manage a translation between the memory-mapped and streaming busses.

Table 4.1: Existing *VILLASfpga* IP cores.

Name	IP package	Interfaces	Vendor	UG
HLS Model	`hls_*`	AS, AM	User	
XSG Model	`xsg_*`	AS, AM	User	
RTDS Interface	`rtds_axis`	AS, AM, I	ACS	
Simple & SG DMA	`axi_dma`	AS, AM, I	Xilinx	[31]
Memory-mapped to Stream FIFO	`axi_fifo_mm_s`	AS, AM, I	Xilinx	[36]
Stream Data FIFO	`axis_data_fifo`	AS	Xilinx	[38]
PCIe Interrupt controller	`axi_pcie_intc`	AM, I	ACS	[33]
AXI4-Stream Interconnect	`axis_interconnect`	AS, AM	Xilinx	[37]
AXI Memory-mapped Interconnect	`axi_interconnect`	AM	Xilinx	[20]
PCIe-AXI bridge	`axi_pcie`	AM, I	Xilinx	[34]
Timer / Counter	`axi_timer`	AM, I	Xilinx	[35]
Software (SW) Reset Trigger	`axi_gpio`	AM, I	Xilinx	[32]

Xilinx provides several datamover IP cores which have been evaluated and compared in their performance.

Direct Memory Access (DMA)

As introduced in section 2.7.1, a DMA controller can be used to offload the task of data transfer from the CPU. A standard DMA controller transfers data from one memory-mapped location to another. However, as stated earlier, in this application a translation of memory mapped data to streaming data is required.

As shown in figure 4.6, the DMA controller is made up of two channels

- The Memory-Map-to-Stream (MM2S) channel reads data from consecutive memory map locations and sends it to the AXI4-Stream link.
- The Stream-to-Memory-Map (S2MM) channel receives data from the AXI4-Stream and writes it to consecutive memory locations.

So both parts of the DMA controller have a streaming and memory map side. Only the direction and therefore the master / slave role of the streaming side is interchanged. The behaviour of a standard DMA controller can be obtained by connecting the streaming sides of the S2MM and MM2S channels together. The memory map side of both channels is realized as a AXI4-MM master. It will directly access the main memory of the host system by using the AXI-PCIe bridge. A significant advantage of the DMA controller is the ability to perform burst transfers. This means that a transfer of bigger data chunks is accelerated because the address information only has to be transmitted once. Xilinx' DMA controller can be adjusted by several parameters which will affect available features and the required FPGA ressources. The designer has the choice between a Micro, Simple or SG version of

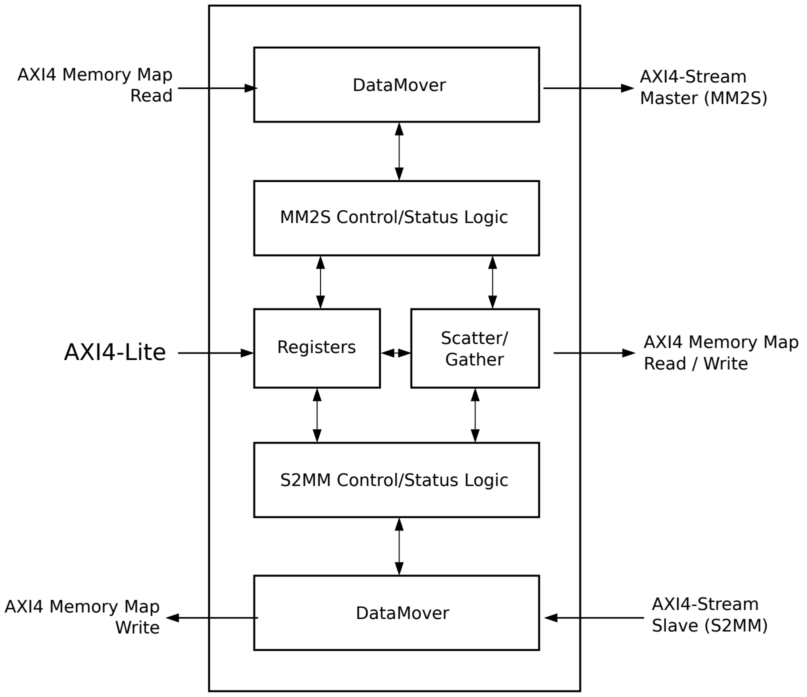


Figure 4.6: Architecture of `axi_dma` DMA controller [31].
.

the IP core. In this implementation both the Simple and SG versions have been used.

Memory-mapped FIFO (PIO)

Another datamover which has been evaluated is based on a FIFO queue.

Xilinx provides two different kinds of AXI4-Stream FIFO's. A simple data FIFO which features both an AXI4-Stream master and slave interface. It is not suitable as a datamover as it lacks the ability to translate between memory map and streaming interfaces. However, it can be used as a buffer between AXI4-Stream nodes which put backpressure on the streaming links. Another type of data FIFO is a memory-mapped FIFO which has an AXI4-Lite interface in addition to the streaming interfaces. This AXI4-Lite interface can be used to fill or read data from the internal FIFO. Figure 4.7 shows two channels similar to the DMA controller.

4.2.2 FPGA Models

Models for *VILLASfpga* can be implemented using Vivado System Generator for DSP (XSG), Vivado High-level Synthesis (HLS) or pure VHDL / Verilog. Each model should have at least AXI4-Stream interface to exchange sample values with other nodes, interfaces or data-movers. Adjustable model parameters can accelerate

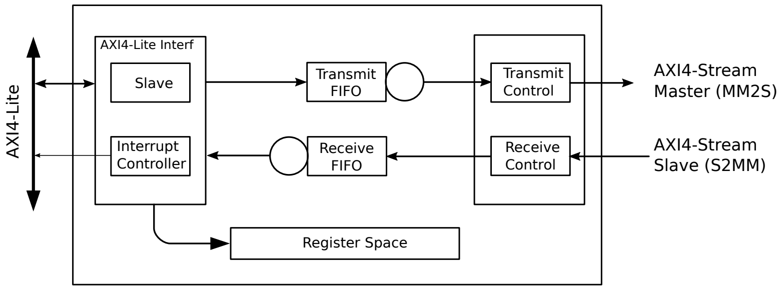


Figure 4.7: Architecture of `axi_fifo_mm_s` FIFO [36].

.

productivity as bitstreams do not have to be regenerated for every change. Such parameters are accessible via AXI4 memory-mapped registers. It is worth mentioning that read / write operations on those registers are not synchronized with processing of AXI4-Stream data.

Both HLS and XSG handle the generation of AXI4-Slave interfaces for parameters automatically. During synthesis, both tools generate a XML-based description of available parameters, their data type, register address and more. This information will be later used by *VILLASnode* to easily adjust parameters.

Vivado System Generator for DSP (XSG)

XSG is a Simulink blockset and a compiler which generates optimized HDL code from models made up of those blocks. In the course of this thesis, simple AXI4-Stream based models have been implemented. For example, figure 4.8 shows a simple pipelined floating point multiplication. A big advantage of XSG is the ability to choose the number format which is used by the FPGA. The designer has the choice between single and double precision IEEE-754 floating point numbers or arbitrary fixed precision data types. XSG also supports multi-rate designs. This allows the designer to partition the model into several synchronous or asynchronous clock domains which run at different frequencies. The support for AXI4-Stream interfaces enables efficient pipelined implementation which can process a new value every couple of clock cycles.

Parameters and other slowly changing values can be set and monitored using an AXI4-Lite interface. An example for such a parameter is the variable factor in figure 4.8. Unfortunately, XSG does not generate a machine readable description of which parameters (Gateway In / Out blocks) are present in the design. Such a description could be used to by *VILLASnode*ăto provide the user an easy way to modify these parameters via the configuration file. To solve this issue, a Matlab-code script has been written which inspects the model an generates such a description. This list of AXI4-Lite parameters including their name, data-type and default value

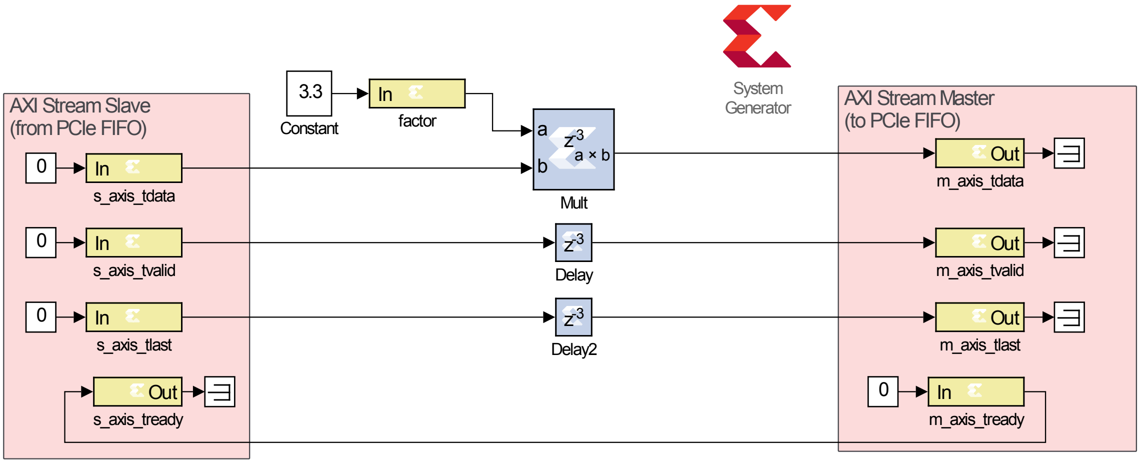


Figure 4.8: Vivado System Generator for DSP model using AXI4-Stream signals.

is then stored in a Read-Only Memory (ROM) which is part of the model itself. During operation, *VILLASnode* reads the description of those parameters from the model also via the AXI4-Lite interface.

Vivado High-level Synthesis (HLS)

Listing 4.1 shows the source code of a model equivalent to the previously presented XSG model. It performs a simple multiplication of 32 bit floating point numbers which are streamed to the `input` port. Like XSG, HLS supports AXI4, AXI4-Lite and AXI4-Stream interfaces for parameters and pipelined streaming data.

To produce efficient code, the designer must follow certain coding style rules. One example is the strictly sequential access to arrays in case a FIFO IP core should be inferred. If the code instead accesses data in a random order, a generic and more costly Block RAM memory will be used by HLS. Furthermore, the user can provide hints to the compiler by using special `#pragma` directives. This approach is already known from other HPC acceleration toolkits like OpenMP or OpenACC.

Xilinx is providing a couple of optimized C++ template libraries which simplify the inferrence of certain IP cores like shift registers or DSP slices. Other examples for such libraries are special classes for arbitrary fixed precision numbers or FIFO's.

HLS has the big advantage that a designer can start from an existing C/C++ implementation and successively apply optimizations to obtain a performant and pipelined HDL implementation. On the path to this goal, the designer will always maintain a standard compliant C/C++ program which can be compiled with standard compilers. By providing test-benches, the user can verify both the compiled software and HDL implementation and compare the results to each other.

Appendix A.2 shows the XML-based accelerator map file which describes the interface of the HLS model. As mentioned earlier, this file includes all details required to control the HLS model later on from *VILLASnode*.

```
#include "ap_int.h"
#include "hls_stream.h"

struct axis {
  float data;
  ap_uint<1> last;
};

void hls_multiply(hls::stream<axis> &input,
   ↪ hls::stream<axis> &output, float factors[32]) {
  #pragma \ac{HLS} \ac{INTERFACE} ap_ctrl_none port=return
  #pragma \ac{HLS} \ac{INTERFACE} axis port=input,output
  #pragma \ac{HLS} \ac{INTERFACE} s_axilite port=factors
     ↪ bundle=config
  #pragma \ac{HLS} DATAFLOW

  static ap_uint<8> i;

  while(!input.empty()) {
    axis in, out;

    input >> in;

    out.data = in.data * factors[i++ % 10];
    out.last = in.last;

    output << out;

    if (in.last)
      i = 0;
  }
}
```

Listing 4.1: Example HLS model

VHDL

The most fine grained control over FPGA ressources are attained when implementing the model in pure HDL. Usually, this is the only possibility for specialized interfaces or in situations where cycle accurate timing is required. Examples are the RTDS interface which will be presented in the next section. This approach is not manageable for larger models as complexity gets easily out of hand. However, optimized custom HDL can be integrated into XSG models using a black-box block.

Appendix A.3 shows the VHDL implementation of a simple model which applies a fixed gain by using a DSP48 slice of the Virtex 7 FPGA.

4.2.3 RTDS Interface

An connection to the RTDS simulator is realized with the GTFPGA interface which has been introduced in section 3.5.2.

The netlist contains a design entity which is called `RTDS_InterfaceModule`. Figure 4.9 shows the IO ports which are provided by this entity.

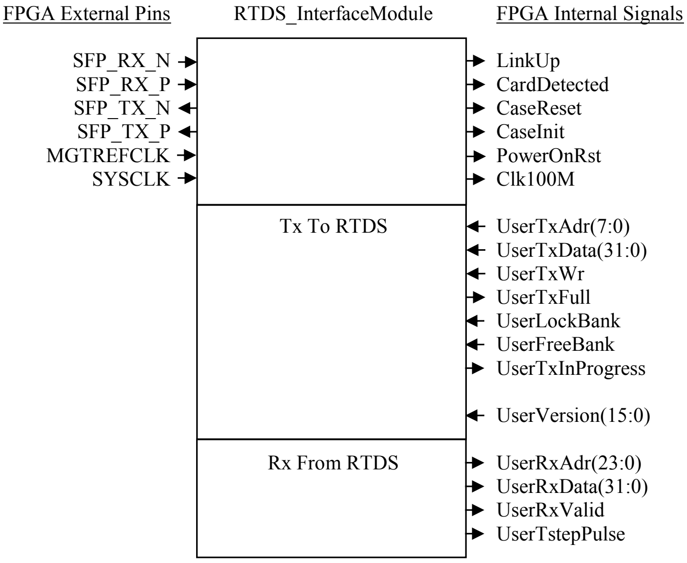


Figure 4.9: The GTFPGA netlist user interface.

External IO interfaces include two clocks and differential send / receive pairs for the SFP cage. The module has to be clocked by a 200 MHz system clock and a 125 MHz clock for the GTX. All signals of the user interface are synchronous to the 100 MHz user clock (`Clk100M`). However, the rest of the FPGA design is based on a clock which is provided by the PCIe component. This requires CDC circuitry to synchronize between the two clock domains.

The GTX block is incorporated into the netlist and drives the SFP module for the optical link. The link itself operates at a link speed of 2.5 Gbit s^{-1} which a 8/10 encoding for error correction. This results in an effective bandwidth of 250 MiB s^{-1}. Every signal is exchanged either as a 32 bit IEEE-754 single precision floating point or integer value. With equation 4.4, a time step of 10uS and 64 values this results in a utilization of around 10 %.

Figure 4.10 shows the timing of user interface signals. Receiving and sending side of the netlist can be accessed independently. Both directions are buffered by internal FIFO queues.

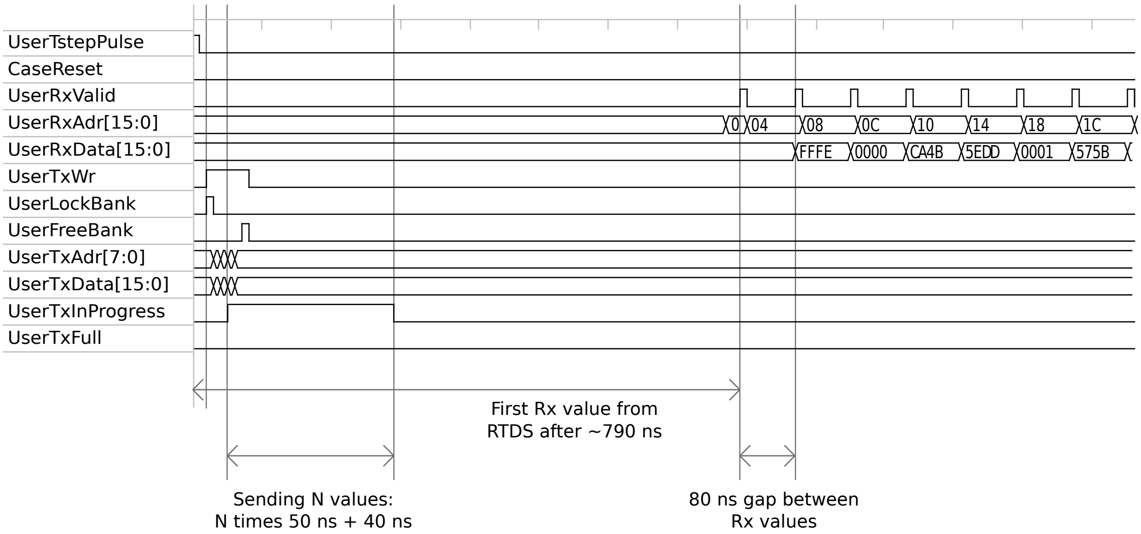


Figure 4.10: Timing wave dump of `RTDS_InterfaceModule`.

The start of a new time step is signalled by RTDS using the `UsertTstepPulse` signal. According to the manual, this signal is delayed by around 400 ns due to latency added by the tranceivers and the optical link.

After the time step has started, user logic can start sending data to RTDS using the state machine in figure 4.11. The user has to provide an address (`UserTxAdr`) and a data word (`UserTxData`) before starting the write transaction by asserting `UserTxWr`. Signals `UserLockBank` and `UserFreeBank` are used to switch between two memory banks in the RTDS processing card. This ensures that all values which are sent will be updated in the same time step.

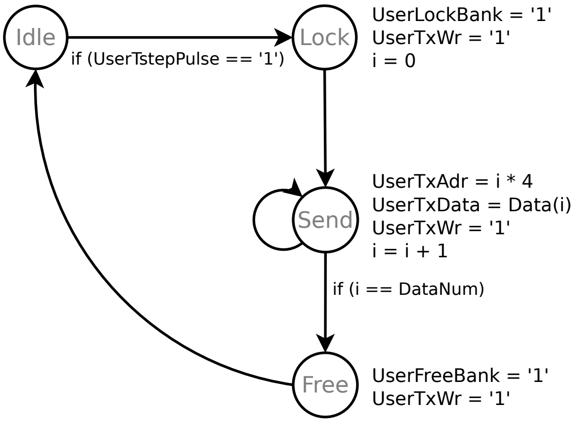
Writing data to the GTFPGA netlist, will enqueue it into an internal FIFO. Transmission is completed as soon as this internal FIFO is drained and signalled by the de-assertion of `UserTxInProgress`. Measurements show that this signal is asserted for $n_{clk,Tx} = n_{values} * 5 + 4$ clock cycles in case n_{values} values have been previously written.

Approximately, 80 clock cycles after the time step has started, the netlist receives the first value from the simulator. This number varies depending on the priority of the GTFPGA block in the RSCAD draft which will affect its scheduling. Every 8 clock cycles a new value is provided to the user logic by asserting the `UserRxValid` signal.

If 64 values are both sent and received to / from RTDS, the last value is received 584 clock cycles (5.84 µs) after the assertion of `UserTstepPulse`.

AXI4-Stream wrapper

To integrate the netlist into the *VILLASfpga* framework, a wrapper is required which connects the user interface of the `RTDS_InterfaceModule` to AXI4-Streaming



1. Wait for: `UserTstepPulse` to pulse for 1 clock cycle
2. Lock Bank: assert `UserWr` & `UserLockBank` for 1 clock cycle
3. Write Data: valid `UserTxAdr` & `UserTxData` with asserted UserWr
4. Free Bank: assert `UserWr` & `UserFreeBank` for 1 clock cycle
5. Wait for: `UserRxValid` to pulse repeatedly until all values have been received (`UserRxAdr` & `UserRxData` valid)

Figure 4.11: State machine for sending values via the `RTDS_InterfaceModule`.

busses. The wrapper has been implemented in VHDL and incorporates a state machine for each direction.

Additionally, an AXI4-Lite interface has been added to provide access to status and control registers. Listing 4.2 shows the register offsets and their function.

```
/* Register offsets */
#define RTDS_AXIS_SR_OFFSET    0x00  /**< Status Register (read-only). See RTDS_AXIS_SR_* constant. */
#define RTDS_AXIS_CR_OFFSET    0x04  /**< Control Register (read/write) */
#define RTDS_AXIS_TSCNT_LOW_OFFSET  0x08  /**< Lower 32 bits of time step counter (read-only). */
#define RTDS_AXIS_TSCNT_HIGH_OFFSET 0x0C  /**< Higher 32 bits of time step counter (read-only). */
#define RTDS_AXIS_TS_PERIOD_OFFSET  0x10  /**< Period in clock cycles of previous time step (read-only). */
#define RTDS_AXIS_COALESC_OFFSET  0x14  /**< \ac{IRQ} Coalescing register (read/write). */
#define RTDS_AXIS_VERSION_OFFSET  0x18  /**< 16 bit version field passed back to the rack for version reporting (visible from ""status command, read/write). */
#define RTDS_AXIS_MRATE     0x1C  /**< Multi-rate register */

/* Status register bits */
#define RTDS_AXIS_SR_CARDDETECTED (1 << 0)/**< ‘’1 when \ac{RTDS} software has detected and configured card. */
#define RTDS_AXIS_SR_LINKUP    (1 << 1)/**< ‘’1 when \ac{RTDS} communication link has been negotiated. */
#define RTDS_AXIS_SR_TX_FULL     (1 << 2)/**< Tx buffer is full, writes that happen when UserTxFull"=1 will be dropped (Throttling / buffering is performed by hardware). */
#define RTDS_AXIS_SR_TX_INPROGRESS  (1 << 3)/**< Indicates when data is being put on link. */
#define RTDS_AXIS_SR_CASE_RUNNING (1 << 4)/**< There is currently a simulation running. */

/* Control register bits */
#define RTDS_AXIS_CR_DISABLE_LINK 0 /**< Disable \ac{SFP} \ac{TX} when set */
```

Listing 4.2: Registers of `rtds_axis`

The wrapper has been packaged together with the GTFPGA netlist into an IP core called `rtds_axis` which can be added to an IPI block diagram.

Three interrupt lines are provided to notify other IP cores or the host CPU about events like the beginning of a time step (`irq_ts` or simulation case (`irq_case`). In case of a running simulation, `rtds_axis` will indicate an overrun by asserting the `irq_overflow` line. This condition is detected when no data has been sent or the transmission is still in progress while the subsequent time step is starting.

The RTDS-to-AXI-Stream interface supports coalescing as described in section 2.7.2. Although it is usually a bad idea to use this feature in a tightly coupled system. It is useful in applications where the interface is just used to capture large amounts of simulation data. In this case the number of generated interrupts can be reduced. Data must then either buffered in FIFO or directly written to the host memory using a DMA datamover. Multi-rate simulations are another use case where coalescing is helpful.

RSCAD

RSCAD is a suite of tools for RTDS simulators. Its *Draft* module provides a graphical user interface to model complex power systems. All of the components described in section 3.5 feature a block which can be instantiated into the design. Figure 4.12 shows the block which is used for the GTFPGA interface.

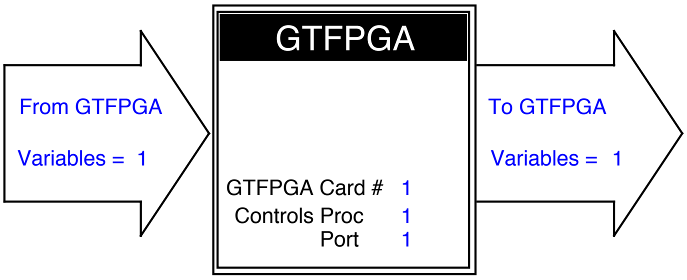


Figure 4.12: RSCAD block for GTNET interface.

Another tool of interest is *CBuilder* which allows to create new blocks for the *Draft* module by using a variant of the C programming language. CBuilder has been used to implement simple models which are used to verify the correctness of the RTDS interface. As the models are implemented in a C-like language, they can be easily ported to Linux. By complying to a basic structure CBuilder control models can be compiled to run on *VILLASnode* without modifications.

4.2.4 Interrupt Controller

Several of the aforementioned IP cores support interrupts. For example, data-movers trigger an interrupt to signalize the completion of a transfer or an error condition. As another example, the RTDS interface generates a new interrupt evertime a new time step begins. These interrupts are a simple signal which is either edge or level sensitive. All interrupt lines are connected to the interrupt controller which is described in this section.

The interrupt controller forwards these interrupts to the host CPU using MSIs. The AXI-PCIe bridge already includes support for sending MSI interrupts to the IO APIC (IOAPIC) of the host system. Up to 32 different interrupt sources can be differentiated by using a 5-bit vector. The delivery of an MSI interrupt is acknowledged by a signal named `INTX_MSI_Granted`. Only after this acknowledgement, the bridge is ready to accept the next interrupt from the interrupt controller.

The interrupt controller in the *VILLASfpga* implementation is based on a Programmable Interrupt Controller (PIC) which is designed by Xilinx for their Microblaze processors. This PIC has a slightly different acknowledgment procedure than the one which is provided by the AXI-PCIe bridge. Hence, the PIC (`axi_intc`) has been encapsulated in a wrapper (`axi_pcie_intc`) which emulates the acknowledgment procedure of the Microblaze processor.

The main task of the interrupt controller is to detect interrupts on up to 32 interrupt lines and to generate the appropriate 5-bit vector when the previous MSI has been acknowledged. In case multiple interrupts are triggered simultaneously, the interrupt controller has to decide which one gets serviced first by assigning priorities. Furthermore, individual interrupts can be disabled or emulated by software for testing.

SW writing to a special interrupt status register can trigger an interrupt manually. This feature has been used to measure the interrupt latency from FPGA to the userspace application.

Table 4.2: Several interrupt sources of the *VILLASfpga* framework.

Signal	**Clock**	**Type**
`irq_timer_0`	`pcie_0.axi_aclk_out`	rising edge
`irq_timer_1`	`pcie_0.axi_aclk_out`	rising edge
`irq_dma_mm2s`	`pcie_0.axi_aclk_out`	level high
`irq_dma_s2mm`	`pcie_0.axi_aclk_out`	level high
`irq_fifo`	`pcie_0.axi_aclk_out`	level high
`irq_rtds_ts`	`rtds_axis_0.clk100m`	rising edge
`irq_rtds_overflow`	`rtds_axis_0.clk100m`	rising edge
`irq_rtds_case`	`rtds_axis_0.clk100m`	rising edge

4.2.5 Timer / Counter

Xilinx provides a timer / counter IP core which can used to periodically generate interrupts or Pulse-width Modulation (PWM) signals. A input capture unit allows measuring the timing of other external signals like interrupts. The main use case in

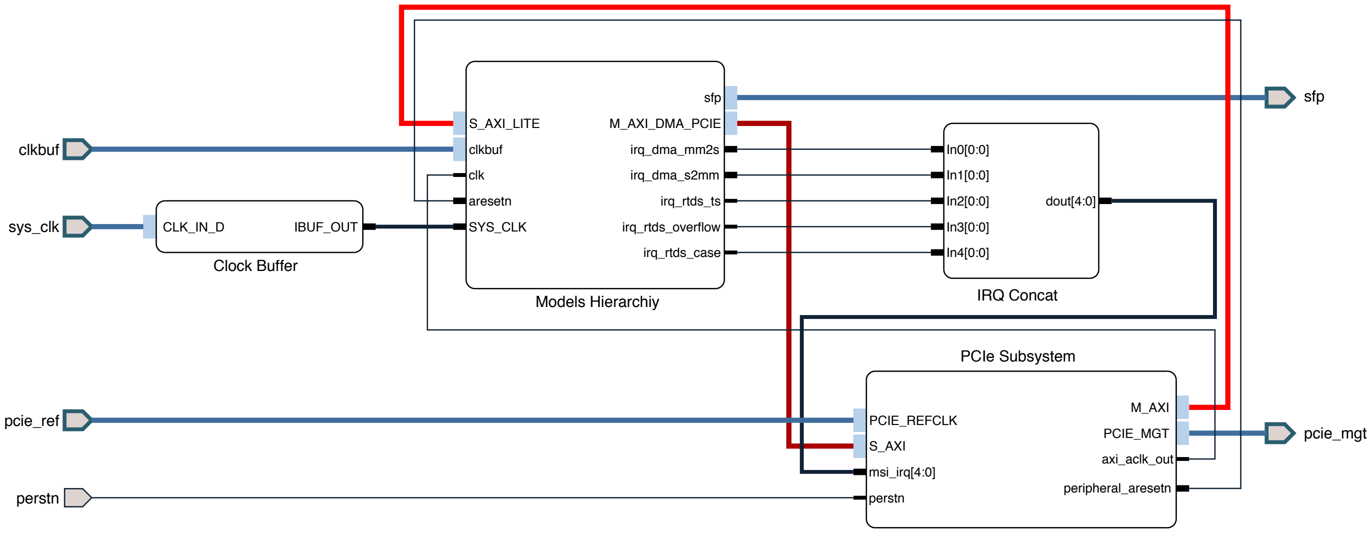


Figure 4.13: IPI: Top-level design.

the scope of *VILLASfpga* is the generation of a periodical time step signals which can be used to synchronize FPGA- and CPU models. Also, it could be used to measure period of an external synchronization signal like the RTDS time step period. In the course of this work, the timer IP core was used to benchmark the synchronization latency and jitter between the FPGA and the host CPU.

4.3 Vivado IP Integrator (IPI)

The Vivado IP Integrator is used to assemble the IP cores into a block diagram. This graphical tool makes it easy to quickly modify certain parts of the top-level design without touching with HDL code. Figures 4.13 show block diagrams of a simple *VILLASfpga* design. It consists of the PCIe subsystem and the model hierarchy which are shown in detail in diagrams 4.14 and 4.15.

Initially, this integration step has be done with Vivado System Generator for DSP (XSG). As XSG is based on Simulink, it offers a similar Graphical User Interface (GUI) like IPI to build block diagrams. However, XSG is mainly designed for implementing DSP related models. It is not well suited for the integration of complex AXI4-based IP cores like the ones listed in table 4.1. IPI supports the integration of XSG as well as HLS models as shown in diagram 4.15. Several limitations of XSG then lead to a re-implementation of the top-level design in IPI which is more flexible and allows finer control over several details of the FPGA design.

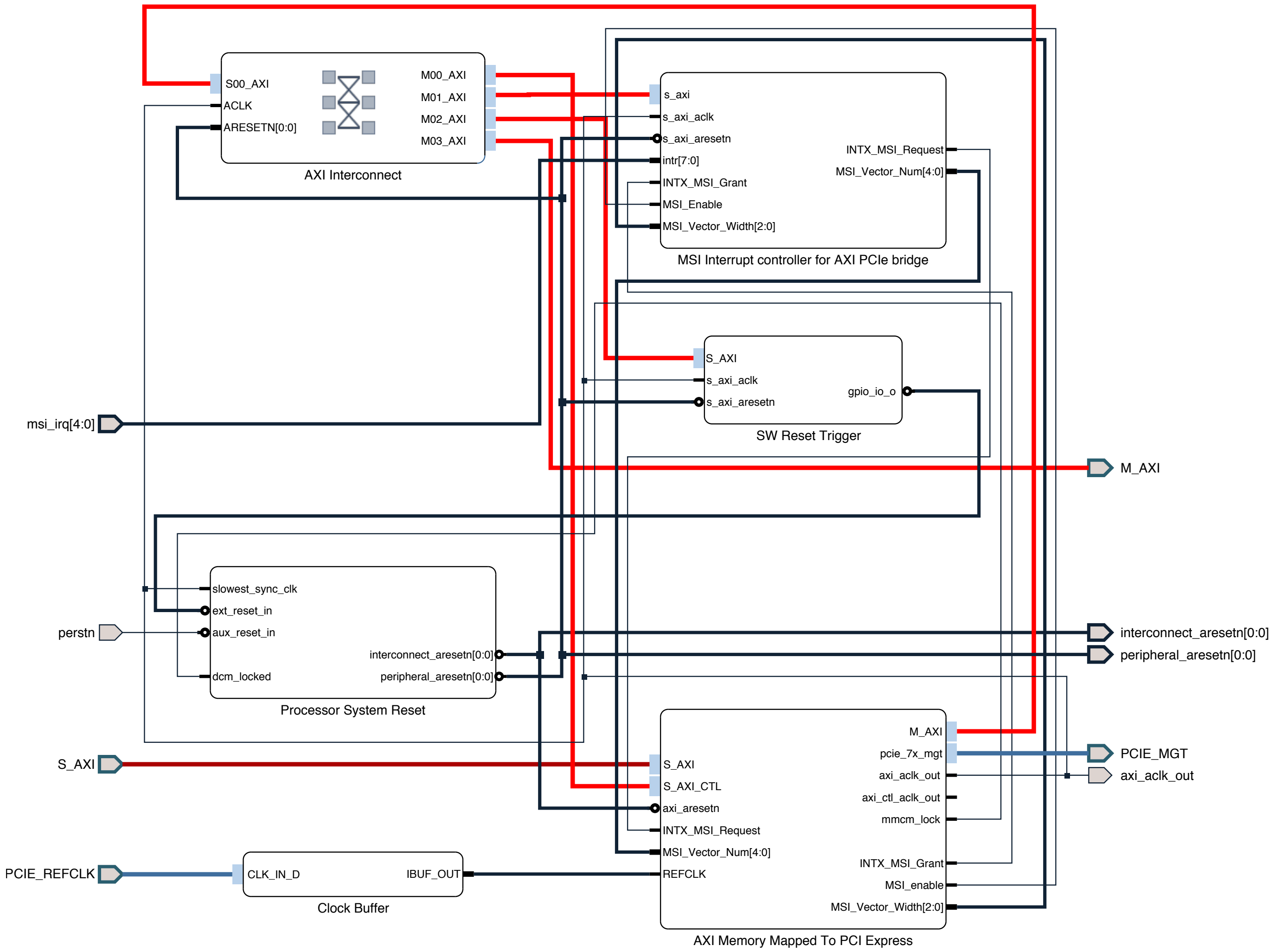


Figure 4.14: IPI: PCIe subsystem.

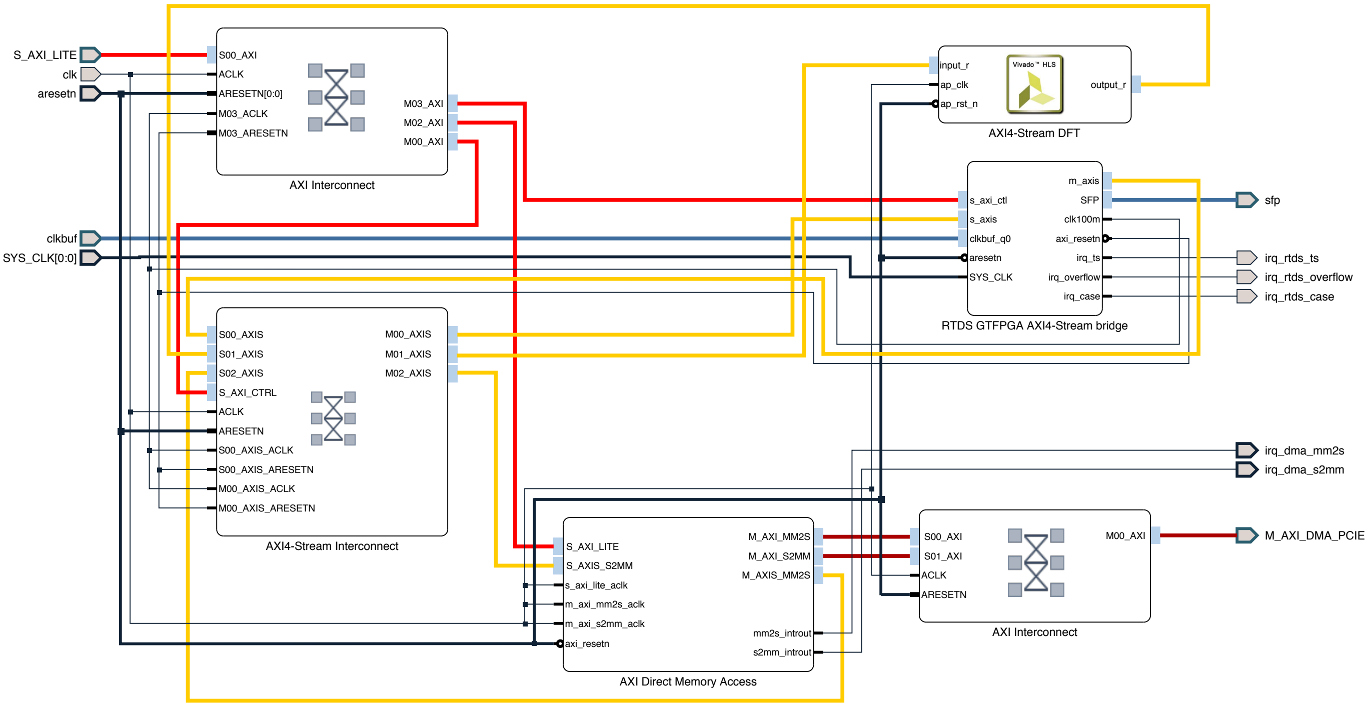


Figure 4.15: IPI: Model Hierarchy.

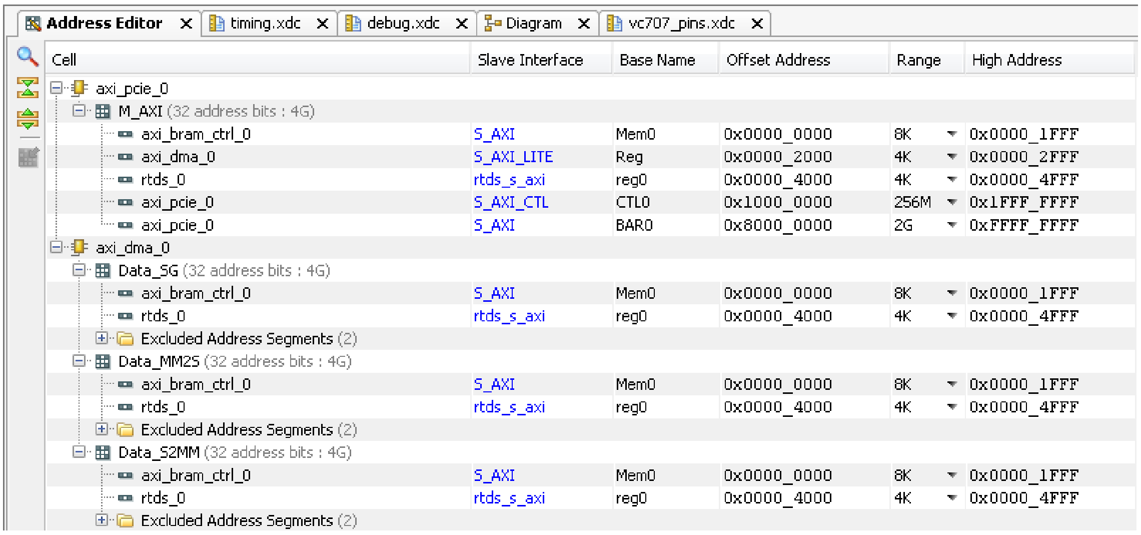
Every IP core which is either connected to a AXI4-Lite or AXI4 bus will need some address space. The address layout of memory mapped busses can be adjusted by the IPI Address Editor as shown in figure 4.16.

4.4 Host Machine

As described in section 3.4.2, the VFIO API is used to access the FPGA card from a user space application. In this case *VILLASnode* is the application which has been extended for this purpose. VFIO imposes several requirements to the hardware of the host machine. To properly isolate the PCIe device an IOMMU must be present in the system. Intel Virtualization Technology for Directed IO (VT-d) provides this feature on newer CPU micro-architectures (Intel Nehalem onwards).

The IOMMU translates memory accesses from IO devices to the main memory. Figure 4.17 shows the relationship to the Memory Management Unit (MMU) which has a similar purpose. Both MMUs can influence certain characteristics of the memory access like caching or prohibit it at all.

Figure 4.18 shows the complete data path and the different address spaces which are traversed. The userspace driver has to take these address translations into account when configuring DMA transfers or accessing memory-mapped registers.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_pcie_0					
M_AXI (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x0000_0000	8K	0x0000_1FFF
axi_dma_0	S_AXI_LITE	Reg	0x0000_2000	4K	0x0000_2FFF
rtds_0	rtds_s_axi	reg0	0x0000_4000	4K	0x0000_4FFF
axi_pcie_0	S_AXI_CTL	CTL0	0x1000_0000	256M	0x1FFF_FFFF
axi_pcie_0	S_AXI	BAR0	0x8000_0000	2G	0xFFFF_FFFF
axi_dma_0					
Data_SG (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x0000_0000	8K	0x0000_1FFF
rtds_0	rtds_s_axi	reg0	0x0000_4000	4K	0x0000_4FFF
Excluded Address Segments (2)					
Data_MM2S (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x0000_0000	8K	0x0000_1FFF
rtds_0	rtds_s_axi	reg0	0x0000_4000	4K	0x0000_4FFF
Excluded Address Segments (2)					
Data_S2MM (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x0000_0000	8K	0x0000_1FFF
rtds_0	rtds_s_axi	reg0	0x0000_4000	4K	0x0000_4FFF
Excluded Address Segments (2)					

Figure 4.16: IPI: Address editor for configuring the AXI4 memory-map.

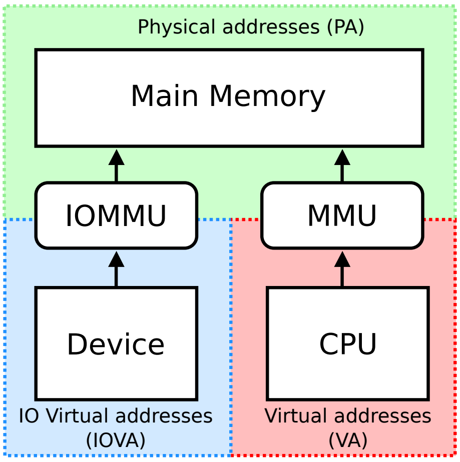


Figure 4.17: Relation of MMU and IOMMU [11].

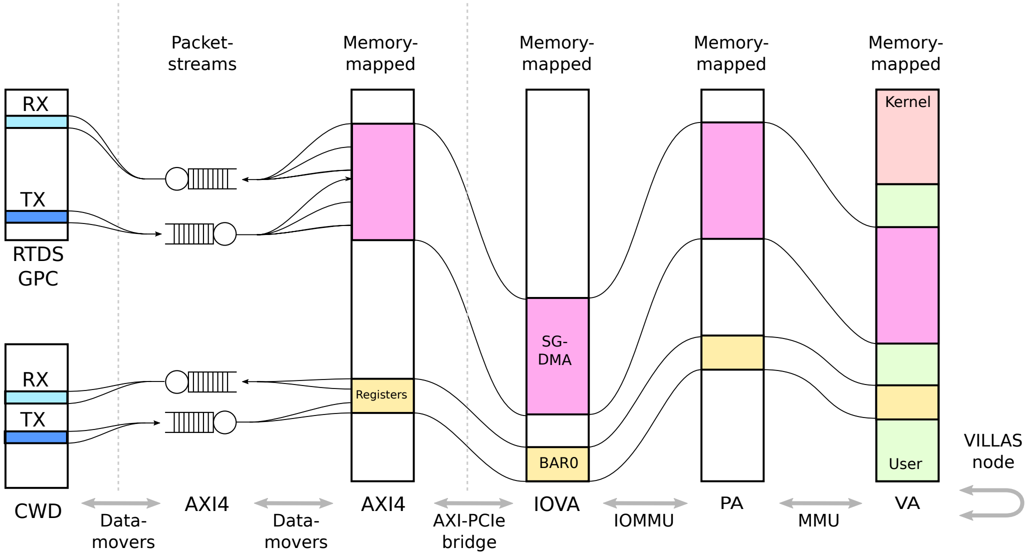


Figure 4.18: Address spaces and their translation involved in the *VILLASfpga* / *VILLASnode* integration.

A fine-grained isolation of certain PCIe cards is made possible with PCIe Access Control Services (ACS). ACS can isolate every PCIe Bus Device Function (BDF) into its own IOMMU group. Therefore every PCIe device will get its own set of translation tables in the IOMMU and therefore gain the ability to fully isolate the devices. Without ACS all PCIe devices connected to the same PCIe root complex will part of the same IOMMU group.

Before a device can be controlled via the VFIO API, all devices which are part of the same IOMMU group must be bound to the VFIO driver. This is a security precaution because devices belonging to the same IOMMU group can still read and write to each others memories. In systems which lack the support for ACS, this would require that all PCIe devices like network or graphics card must be bounded to the VFIO driver and therefore loose their original functionality.

4.4.1 Linux Real-time optimizations

Linux is not a real-time OS by default. However, several optimizations can be made to get almost hard real-time qualities.

The `PREEMPT_RT` patch-set is an ongoing effort to improve real-time properties of Linux [1]. Several contributions of this patch-set set have been already merged to the mainline kernel. `PREEMPT_RT` makes the kernel space preemptive. Hence, operations like network or disk IO which are processed by the kernel can be interrupted by other processes which have higher priorities. This has been made possible by replacing most spinlocks in the kernel with mutexes that support priority inheritance, as well as moving all interrupt and software interrupts to kernel threads.

[1] `https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch`

System latency is a critical parameter for hard real-time applications. It measures the maximum time the user task can be interrupted by other tasks or the OS itself. Unexpected and high system latencies cause the application to miss its deadline. In regard to *VILLASfpga* this means that results for the next time step will not arrive in time at the destination simulator. Usually this simulator will then fall back to use the results of the previous time steps which decreases simulation fidelity and in the worst case causes system instabilities. Main causes for these latencies and a pervasive OS noise are non-maskable interrupts, non-uniform memory and IO access or contentions of system busses [29]. Most but not all of these disturbing side effects can be isolated to certain *noisy* CPU cores while others can be used for the execution for hard real-time tasks.

We can distinguish three main classes of interrupts:

1. The local timer interrupt periodically invokes the OS's scheduler. The scheduler checks if there are any other tasks currently waiting to be executed and eventually performs a context switch to one of those if they are of higher priority or the quota of the current process expires. On a multi-core machine, certain CPU cores should be isolated for the exclusive use by *VILLASnode*. By using the kernel boot parameter `isolcpus=2,3`, the OS is prohibited to automatically schedule other processes to those cores. If there are no other runnable tasks on the core, the schedule will just do some housekeeping and pass control back to *VILLASnode*. Even though just for a small amount of time, *VILLASnode* would still be interrupted periodically. Newer Linux version can disable this periodic timer interrupt and run in a so called *tickless* mode. The corresponding kernel boot parameter is `nohz=on nohz_full=2,3`. Special attention is to be paid if a tickless kernel is used together with the `PREEMPT_RT` patchset. Since version 4.0 of the Linux kernel, the real-time optimized version can not completely disable the local timer interrupt [2].

2. Devices interrupts like MSIs, mice, keyboard or NIC will cause the OS to pass control to the device driver which is in duty to handle the interrupt condition. Using the file `/proc/irq/[id]/smp_affinity` each Interrupt Request (IRQ) can be pinned to a set of CPU cores. This feature can be used to move all non important IRQs away from the isolated cores. Devices which are not used by the application can be completely disabled in the system firmware (BIOS) or by blacklisting the device driver in the kernel. Of course, certain IRQs should be pinned to the cores which are used by *VILLASnode*. Examples are the interrupts of the *VILLASfpga* card or NICs which will be used for communication.

3. System interrupts like NMI, SMI or various exceptions are of special interest because they can not be masked[3]. Exceptions like a division by zero or

[2] http://www.spinics.net/lists/linux-rt-users/msg14226.html

[3] An interrupt can be temporarily disabled by masking it.

a segmentation fault only occur due to erratic program behaviour. It is the responsibility of the application programmer to prevent them by adding appropriate tests. The page fault exception is a special case because it is also triggered under normal operating conditions. Examples are memory accesses to memory areas which have been allocated by a user application but never actually used. Linux delays the actual allocation of memory until it is touched for the first time. Another situation in which Page Fault (PF) exceptions are thrown is in case of memory shortage. The OS will swap certain memory regions out of the memory and store them on the hard drive. Whenever the user access such an area, the OS has to load the previously swapped-out region back to memory. As this can also happen during the normal execution of a real-time application, special care has to be taken by *locking* and *pre-faulting* all memory.

Unfortunately, some interrupts like System Management Interrupts (SMIs) or Non-maskable Interrupts (NMIs) are even out of the control of the operating system. The SMI is used to perform tasks like fan control or power management and can suspend the whole system between micro to milliseconds. The routine which handles this interrupt is part of system firmware and completely out of control of the OS. Hence, it can be a killer of real-time applications. Tools like `hwlatdetect` from the `rt-tests` suite[4] can be used to detect the existence of SMIs and to measure their maximum latency.

The number of interrupts which interrupt a certain core can be monitored by `/proc/interrupts`. This is an easy way to check the effect of the previous optimizations.

Further optimizations can be done in the system firmware (BIOS). All devices which are not required by the application should be disabled or physically removed from the machine. Switching the disk controller from Advanced Host Controller Interface (AHCI) to AT Attachment (ATA) mode can improve the system latency. Because of possible SMIs, all features related to power management, adaptive clocking (Intel speed step), support for legacy Universal Serial Bus (USB) inputs devices or fan control should be disabled. Intel's virtualization technology (VT-d) is a requirement to use VFIO in the userspace application.

A tool called *tuned*[5] should be used to optimize the runtime kernel configuration in regard to real time. By default Redhat based Linux distributions ship *tuned* with a real-time profile which already includes most of the optimizations which have been described above.

Another good resource for building real-time optimized applications is the Linux RT-Wiki[6].

[4] https://git.kernel.org/cgit/utils/rt-tests/rt-tests.git/

[5] https://fedorahosted.org/tuned/

[6] https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application

4.4.2 Userspace Drivers

Section 3.4 covered two Linux userspace APIs for accessing PCIe devices, namely UIO and VFIO. For the integration of *VILLASfpga* and *VILLASnode*, the VFIO API has been chosen due to its range of features. VFIO provides an API to access the PCIe card by means of MMIO. This means that parts of the AXI4 address space are mapped into the Virtual Address (VA) space of the user space application.

Most of the previously described IP cores have a dedicated AXI4-Lite interface which is used to access registers of this core. These registers are then used to control the components.

Xilinx provides drivers for most of the IP cores[7]. They provide an abstraction layer on top of direct register access. Technically, these drivers are intended to be used with applications which run ARM or Microblaze processors. These drivers are designed to be used in standalone applications and therefore to not pose any requirements on an underlying OS. This makes it really simple to integrate them in the *VILLASnode* userspace application.

For HLS and XSG models, drivers are automatically generated by Xilinx tools as part of the synthesis process.

4.5 Synchronization

There are two fundamental different ways to synchronize the host system with the FPGAs. Both of them have been described in section 2.7.2.

Interrupts are delivered by the previously described interrupt controller by means of MSIs. MSIs are automatically acknowledged by the OS and the host system interrupt controller.

VFIO will forward the interrupt requests to the user application by using a so called event file descriptors (*eventfd*). This special file descriptor will block on a `read()` operation until the next event occurs. Alternatively common file descriptor operations like `poll()` or `select` are supported.

This requires that every interrupt is first handled in the VFIO kernel module before it is passed to the user application. Fortunately this has the advantage that the kernel can count the number of interrupts which occurred and report this to the user application. So in case the user application is busy, a missed interrupt can be detected.

The alternative polling does not require interaction with the VFIO kernel module. Most IP cores and the PIC itself have an interrupt status register. By disabling the physical assertion of the interrupt line, this status register can be used to detect an interrupt condition by repeatedly reading its value via PIO.

[7] `https://github.com/Xilinx/embeddedsw`

4.6 Design Flow

This chapter proposes a design flow which can be used to extend the *VILLASfpga* framework. The description of those steps in a flow is a common methodology for Application-specific Integrated Circuit (ASIC) and FPGA design. The variety of design entry methods, intermediate files and tools justify this formalization. Figure 4.19 illustrates the flow on the whole.

Like its software counterpart, *VILLASfpga* can be extended with additional node-types. A node-type is either a model or an interface. Those node-types can be implemented by three different methods. Examples for custom VHDL code, Simulink-based models and C/C++ code are given in A.

Simulink models are compiled by the Vivado System Generator into VHDL or Verilog code. Likewise, C / C++ code is compiled by Vivado's High-level Synthesis tool. The generated or self-written HDL code has to be packaged by Vivado into a IP package. Packaged IP is managed by Vivado's IP Catalog. This repository of contains both custom designed node IP and included vendor IP from Xilinx.

The IPI provides a user-friendly GUI for connecting blocks and assigning address space in a block diagram. The diagram is the top-level design entity of *VILLASfpga*. It includes several standard IP blocks from Xilinx like the PCIe-AXI bridge, interrupt and DMA controllers. The user can extend this block diagram with additional node or interface IP which has been packaged in a previous step. New IP has to be connected by Xilinx AXI infrastructure IP.

The block diagram (.bd) is a XML file which is used to generate several output products:

1. A *Hardware Hand-off* specification of included components, their connections and most importantly a memory map. This XML-file (.hwh) will be later used by *VILLASnode* to configure and control the FPGA card.

2. A VHDL / Verilog top-level entity which is used for the integration of all blocks during synthesis.

3. Descriptions of used IP blocks and their parameters in form of XCI files.

Similar to the Hardware Hand-off file, Vivado HLS generates a *Accelerator Map* XML-file which contains a description of parameters, inputs and outputs of the HLS model. Likewise the System Generator Simulink model (.mdl) is a XML-file from which input and output gateways can be extracted. In summary, *VILLASfpga* will use a collection of XML-files to pass a detailed description of the FPGA design to *VILLASnode*.

This design automation simplifies the integration of *VILLASnode* and *VILLASfpga*. Available nodes, interfaces and data-movers on the FPGA are automatically detected by *VILLASnode*.

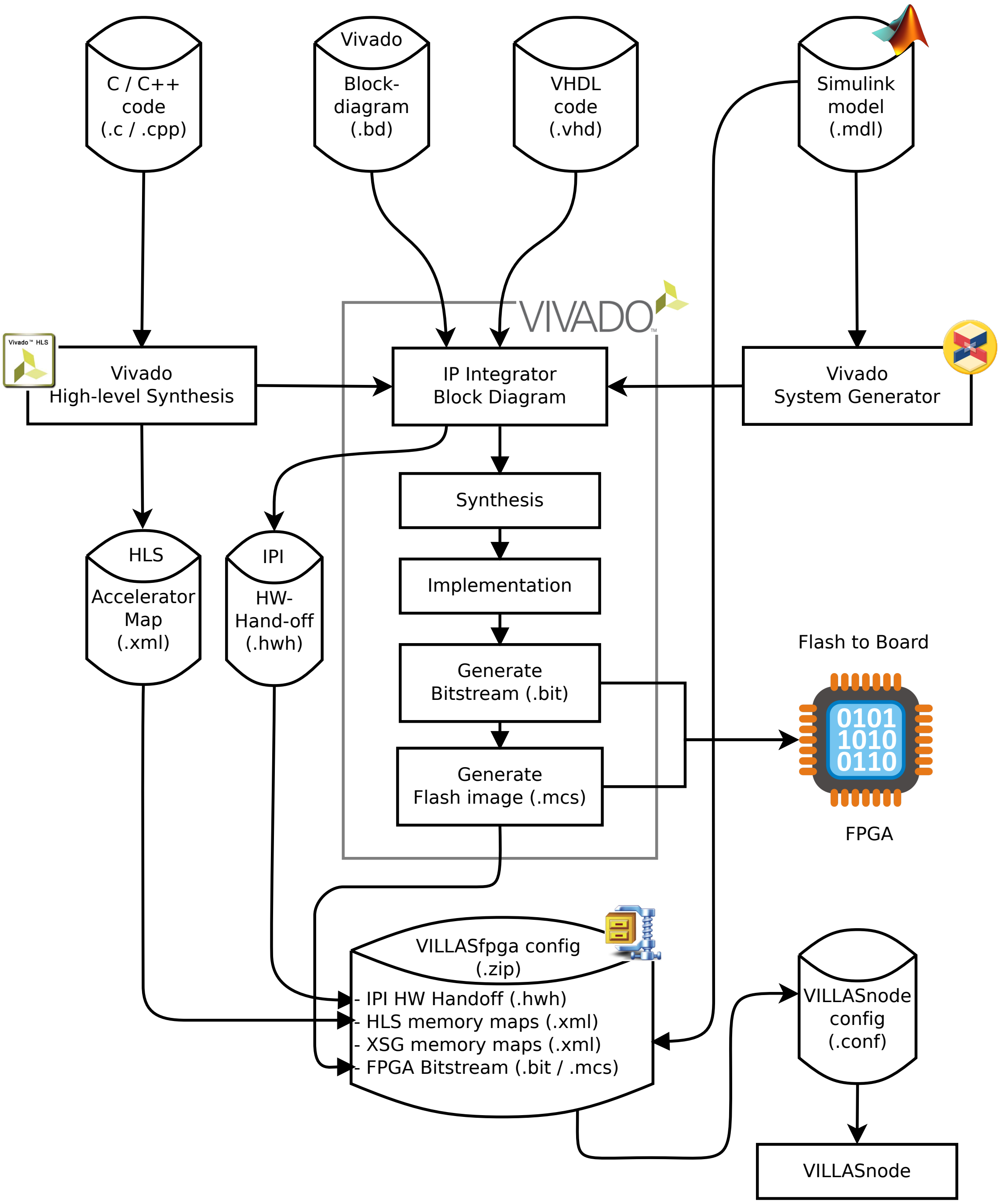


Figure 4.19: Design flow for *VILLASfpga* framework.

After the final integration of all components has been completed, Vivado will be used to start the synthesis and implementation runs. The final result of the implementation is a bitstream file (.bit) It can be directly loaded to the FPGA board using a USB cable and Vivado's hardware manager.

Alternatively, the bitstream can be converted into a MCS file. MCS files describe the data layout for Programmable Read-only Memory (PROM) / flash chips. Those memory chips are used to automatically configure the FPGA without the need for a dedicated workstation. The MCS file only has to be flashed once to the memory chip.

For controlling the *VILLASfpga* board, *VILLASnode* requires detailed knowledge of the actual IP cores which are instantiated in the FPGA design. This includes, the memory map of the AXI busses, interrupt vectors associated to each IP core and the routing of AXI4-Streaming busses between them. This information can be extracted from various side products of the aforementioned flow (Hardware Hand-off file, Simulink Model, HLS Accelerator Maps). Ideally, *VILLASnode* would access those files directly. At the moment, the user must extract this information himself and add it to the *VILLASnode* configuration manually. An example for such a configuration file can be found in appendix A.1.

4.6.1 Software

The *VILLASfpga* framework integrates several design tools. Table 4.3 summarizes the software which has been used.

Table 4.3: Software and Tools.

Product	**Vendor**	**Version**
`RTDS_InterfaceModule` netlist	RTDS	Revision 8 for VC707
RSCAD	RTDS	4.007
Linux Kernel	Linux Foundation	4.4.9 (`PREEMPT_RT`)
Fedora Server Distribution	Fedora Project	23
Vivado	Xilinx	2016.1
Vivado HLS	Xilinx	2016.1
System Generator for DSP	Xilinx	2016.1
Matlab / Simulink	Mathworks	2015a
VILLASnode	ACS	Git

5 Results

The findings of this work are summarized in two separate sections. The first section covers the performance of the PCIe interface between the FPGA and the host system. Afterwards two example applications in the field of power system simulation are presented and evaluated.

Nonetheless, the principal products of this work are design files and software of the *VILLASfpga* framework. FPGA design files and example models are provided within the *VILLASfpga* Git repository[1]. The software which is required to control the FPGA board is integrated into *VILLASnode* and therefore part of its Git repository[2].

5.1 Test System

The following tests have been conducted on an existing server machine which was previously used for *VILLASnode*. All of the real-time optimizations described in section 4.4.1 have been applied. In some benchmarks a standard kernel and a `RT_PREEMPT` kernel are compared. Benchmark processes are always pinned to an isolated CPU on which they are the only runnable task. Interrupts of noninvolved devices are mapped to remaining cores.

Details of the hardware and the software are given in table 5.1.

Table 5.1: Test system for *VILLASnode* benchmarks.

CPU	Intel Xeon E3-1240 V2 @3.40GHz[3]
Chipset	Intel C202[4]
RAM	8 GiB, DDR3 1333 MHz, ECC buffered
Mainboard	ASUStek P8B-C Series[5]
Network	Quad-port Intel 82574L Gigabit Ethernet, onboard[6]

[1] https://github.com/RWTH-ACS/VILLASfpga

[2] https://github.com/RWTH-ACS/VILLASnode

5.2 Interface

5.2.1 PCIe Interface

DMA vs PIO

Out of the three tested data-movers (FIFO, DMA & DMA with SG), the DMA controllers offer the highest bandwidth between the host memory and the FPGA for larger transfers. This is due to the fact that the FIFO datamover is accessed via PIO. Figure 5.1 shows the time required by the CPU to read and write data via PIO to the BAR0 memory region of the FPGA. Scaling is linear as every single data word is transferred in an independent PCIe bus transaction. Especially reading is very costly as every transaction requires the FPGA to generate a completion TLP which contains the requested data. By using Vivado's integrated logic analyzer, the time between two consecutive reads has been measured to about 115 clock cycles (around 1 µs). In comparison, consecutive write operations have a gap of about 5 clock cycles (40 ns). A write transaction is a so called *posted* operation which is not acknowledged by a completion and therefore can be performed faster. Interleaving reads and writes are even more costly as the CPU has to flush its write buffer before the next read can be performed.

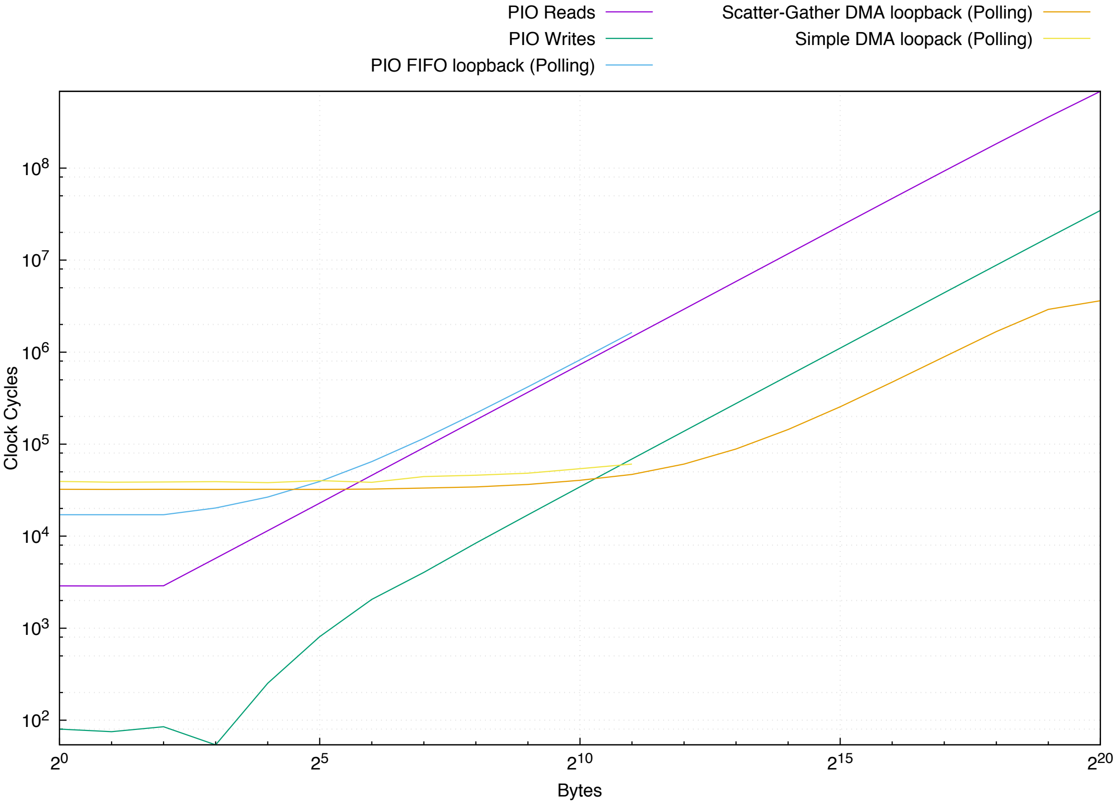


Figure 5.1: Comparison of data-movers and simple PIO read / writes.

Even though, modern Intel x86 CPUs are capable of issuing burst transfers on the PCIe bus, this requires the memory of the FPGA to be mapped with the Write-combine (WC) flag [28]. In this case up to 64 B can be written in a single burst transfer by using AVX2 vector instructions. However, the VFIO driver API can not map the FPGA memory with the WC flag. Similarly, burst read transfers are only possible if the BAR0 is mapped as pre-fetchable.

Due to these limitations, PIO can not be used to transfer data in bursts to the FPGA. PIO-based data-movers will not achieve satisfactory results until further optimizations such as custom kernel modules will enable burst transfers.

Luckily, DMA transfers between the FPGA and the main memory of the host system are not affected by this issue. Every packet on the AXI4-Stream bus is transferred via an AXI4 / PCIe burst transfer. This allows the DMA controller to almost fully saturate the AXI4 busses and reach the theoretical peak bandwidth. It is worth mentioning that this peak bandwidth is much higher than what is required by common *VILLASfpga* applications.

Interrupts vs. Polling

Figure 5.2 compares response time for events delivered with PCIe Message Signalled Interrupt (MSI) versus polling. The benchmark has been conducted by faking an event in the interrupt controller using so called software interrupts. This can be done with a single PIO write to a register of the interrupt controller. Results show that the polling latency averages around 1 µs which is almost half the latency seen with interrupts (> 2 µs). Furthermore, the jitter associated with it is much higher for interrupts. The latency measured for PIO-based polling pretty much matches the time required for a 32 bit PIO read as shown in figure 5.1.

Figure 5.3 shows the jitter of a periodic timer interrupt which is generated on the FPGA. The timer event is captured by the user application either by using polling or MSI.

The results of this measurement are surprising. Usually, one would expect that polling on a quiet real-time system yields the best results. However, the opposite is the case: interrupts on the non real-time system show the smallest amount of jitter. One explanation for this behaviour could be the fact that the benchmark is the only process which runs on that core. There is no scheduling, priority inversion or competition on shared resources in which the real-time kernel could show its benefits. Or it could be that the Linux kernel schedules the interrupt handler based on its own periodic timer interrupt. In this case, the observed jitter would not reflect the jitter of the FPGA timer, but the one of the local APIC timer.

The jitter observed when polling on the interrupt registers is similar for both kernels. However, its deviation is much larger than originally expected. It could be ascribed to the observed gap time between two PIO reads of about 1 µs. This can

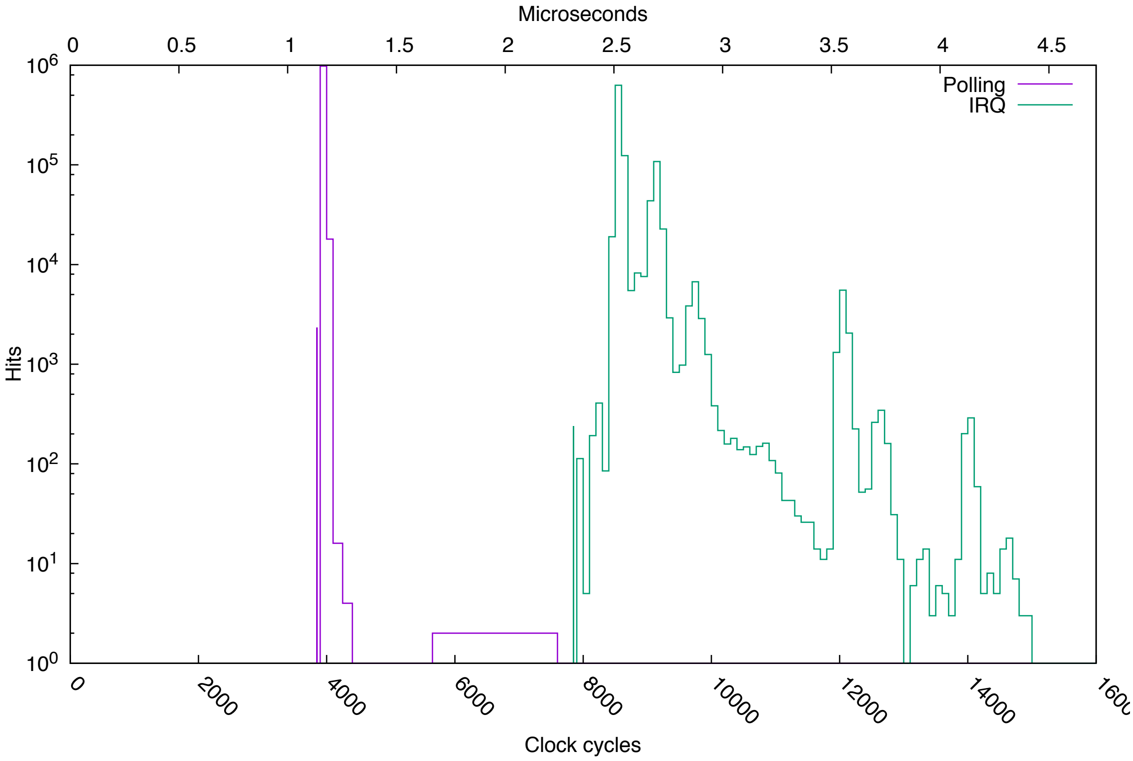


Figure 5.2: Latency of interrupts versus polling for events generated by the FPGA.

be explained with the slow read speed. The polling jitter almost matches the time which is required for a read.

5.2.2 RTDS Interface

Simple Round-trip Time (RTT) measurements have been conducted to measure the latency introduced by the RTDS and PCIe interfaces. Both Linux and the simulator are sending their current time step counter and loopback the counter value of their opponent. The FPGA is just used to forward the simulation data between the two nodes. The RSCAD model shown in figure 5.4 subtracts the received value from the current to precisely determine the actual RTT. Figure 5.5 shows the relationship between the RTTs and the employed communication pattern which was described in section 2.6.

Linux is very flexible in choosing the point in time when data is sent. The best RTT result can be achieved if RTDS sends its counter and Linux directly sends it back. In case neither Linux nor RTDS perform other calculations, the RTT is equal to a single time step. In contrast, RTDS can always send data shortly after the beginning of its time step. It also requires one additional time step (green line) to update the internal control variable before it can be sent back. This limitation leads

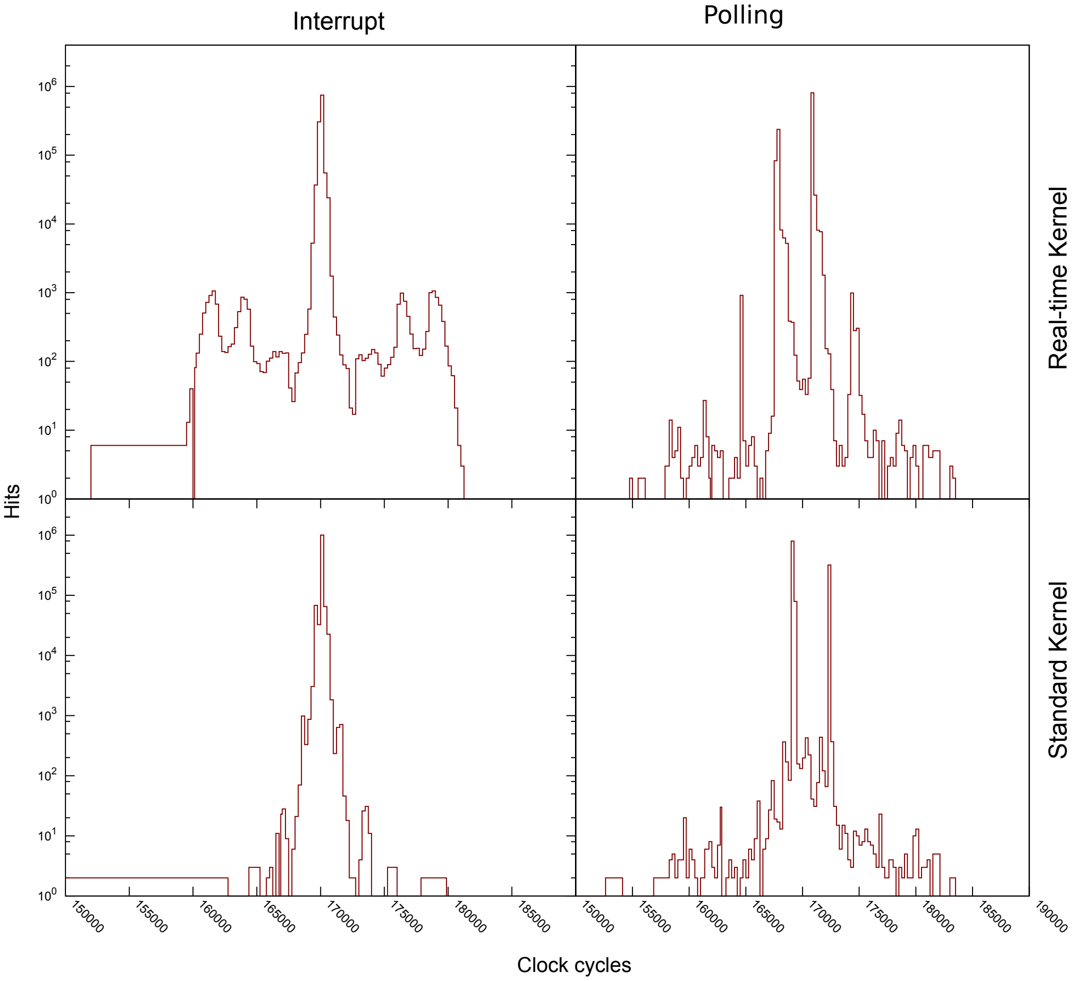


Figure 5.3: Jitter of periodic event generated by the FPGA.

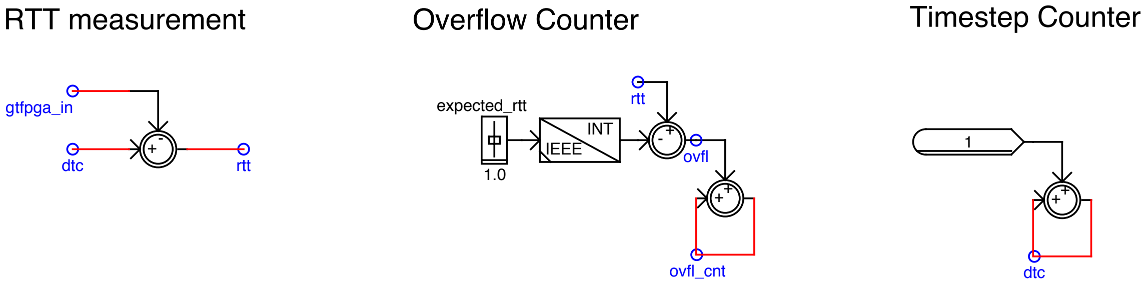


Figure 5.4: RSCAD draft for RTT tests.

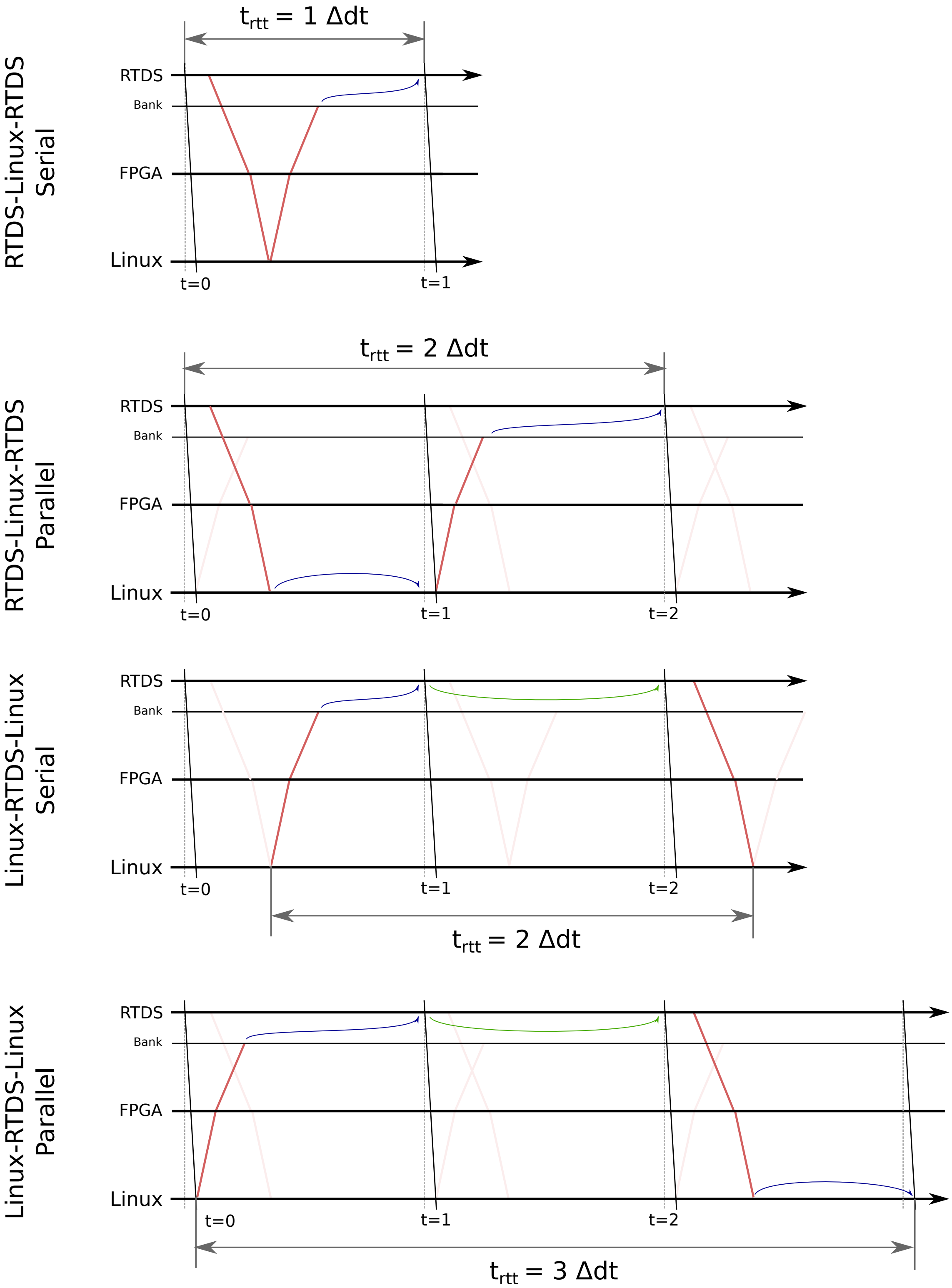


Figure 5.5: Round-trip Time measurement between RTDS and host CPU.

to a RTT of at least two time steps. As a result, RTT is asymmetrical and can not easily be used to derive the one-way latency. However, it helps in understanding the exact timing and restrictions imposed by RTDS.

An overrun caused by the interface occurs in the case when the exchanged signals arrive too late at the destination simulator to be considered in the next time step. This situation can be easily detected by deviation in the measured RTT. The stability of the interface depends on a variety of factors:

1. At first, a distinction based on the communication pattern is required. The parallel pattern decouples the data exchange from the computation because both can be executed concurrently. In the serial case, computation directly reduces the time which is left for communication. Therefore, the parallel pattern can be used with smaller time steps while still having the full time step available to carry out computations. This comes at the cost of an additional time step latency in comparison to the serial pattern. In the serial case, results will be sent back much later (after computation has been finished). This makes it more susceptible to overruns as the period where interrupts can happen is larger and their consequences are more severe.

2. Secondly and much more important is the time step period of the simulation. Tests have shown, that time steps above 25 µs are rather safe in both cases. RTTs have been measured for hours without overruns. Below 25 µs, system latencies caused by interrupts and other factors start to affect the stability. In comparison, OPAL-RT's eMegaSim simulators which use a similar architecture support time steps as low as 20 µs. This simulator is also based on a x86 Linux machine paired with a Xilinx FPGA board which is connected via PCIe. However, the typical time step for EMT-based simulation is 50 µs. If smaller time steps are required, models can be ported to run on the FPGA with timesteps smaller than a micro second.

3. The number of values which are exchanged can have an influence in case that a PIO based data-mover like the FIFO is used. As stated earlier, PIO reads require about 1 µs per signal.

Previous tests did not involve any heavy computation on the Linux machine. Yet the amount of calculations which can be performed without missing deadlines is of interest. To examine the limitations of the interface, the next benchmark solves matrix inversions with varying dimensions. The workload is composed of a LU matrix factorization with row pivoting implemented by Net-lib's highly optimized LAPACK routines. This workload as it is the dominant part in most Nodal-based solvers for power system simulation.

Figure 5.6 shows the results of this test for a time step of 50 µs and a parallel communication pattern. Problem sizes up to $N = 36$ can be solved on the Linux machine without missing any deadlines. Above that, almost all deadlines are missed which will result in an increased latency and instabilities in the system.

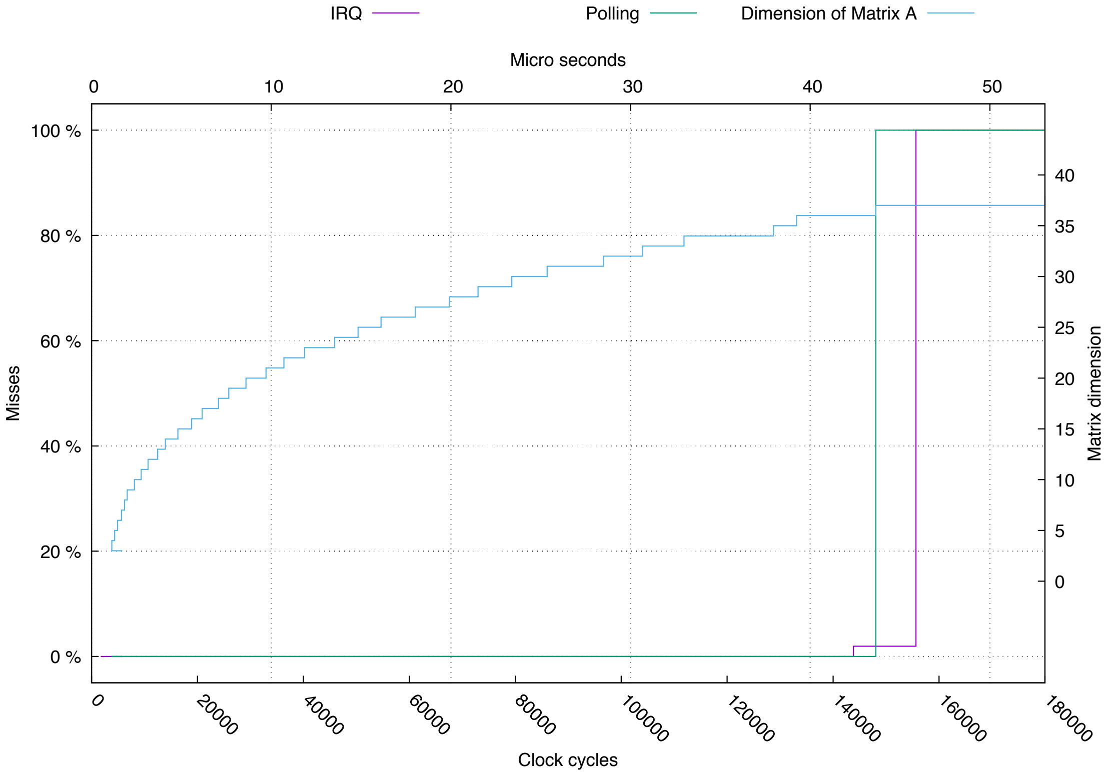


Figure 5.6: Missed deadlines versus dimension of matrix A (dt = 50 µs).

5.3 Applications

5.3.1 Simple Circuit

The RTT tests in the previous section only used control components in the RSCAD model. They did not imply a power system solution. To examine the interplay between the GTFPGA block and a power system solution a simple electrical circuit is used. Figure 5.7 shows the system consisting of simple RLC components and how it has been gradually decoupled into two subsystems SS1 and SS2. The first part of the figure shows the integrated solution. In the second sub-figure, the system is decoupled by an Ideal Transformer Model (ITM). In the next step, SS2 has been replaced by a CBuilder component which models the behaviour of that part of the circuit in C code. The CBuilder control component is not integrated into the power system solution and therefore includes a simple trapezoidal solver. Then same C code has been compiled as a model which is executed by *VILLASnode* on the Linux system. The interface between the RTDS simulator and *VILLASfpga* is handled by the GTFPGA block. All four implementations of this circuit are simulated in parallel to allow for a comparison of the results which are shown in figure 5.8.

5.3.2 Hybrid DP-EMT simulation

To provide an example for the flexibility and performance of the *VILLASfpga* framework, a simple EMT-to-DP transformation has been implemented on the FPGA. The transformation from time to time-frequency domain is performed by a simple short-time DFT which is applied to a sliding window which spans over a period of the input signal.

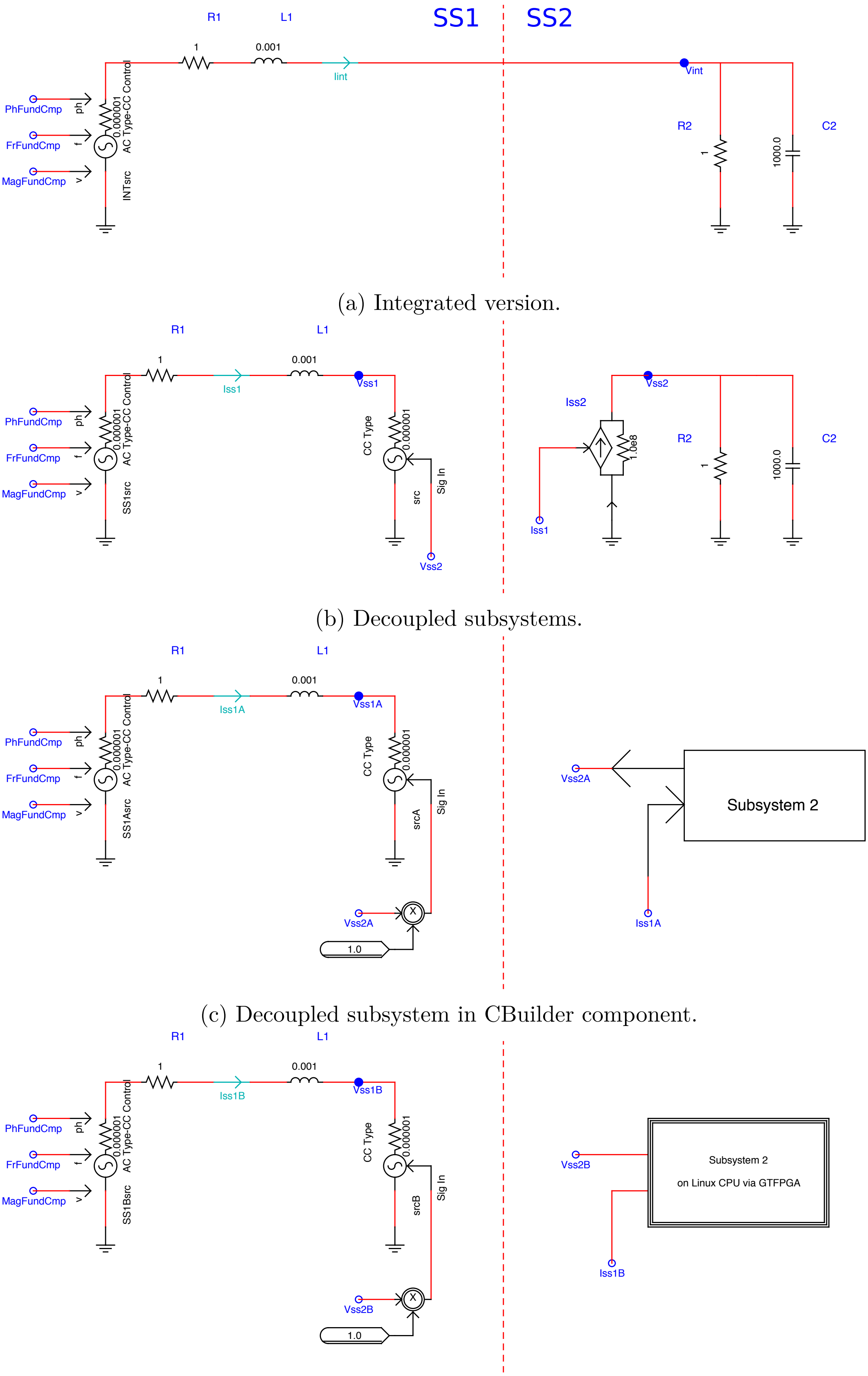
There are efficient recursive algorithms for the calculation of such a DFT if only a couple of frequency bins are of interest [13, 12]. Figure 5.9 shows the transfer function of such an algorithm. As the window is shifted over the signal, the complex DFT coefficients rotate with their corresponding frequency. However, DP are based on a fixed reference frame. Therefore all coefficients must be corrected with a factor $C_k(n)$ in equation 5.3.

$$S_k(n) = e^{j2\pi k}[S_k(n-1) + x(n) - x(n-N)] \tag{5.1}$$

$$C_k(n) = e^{-j2\pi k(t-(N+1)))} \tag{5.2}$$

$$DP_k(n) = \frac{2}{N} C_k(n) S_k(n) \tag{5.3}$$

To showcase the most novel technology, the algorithm was implemented with Vivado High-level Synthesis (HLS). Listing A.4 shows the source code of the transformation.



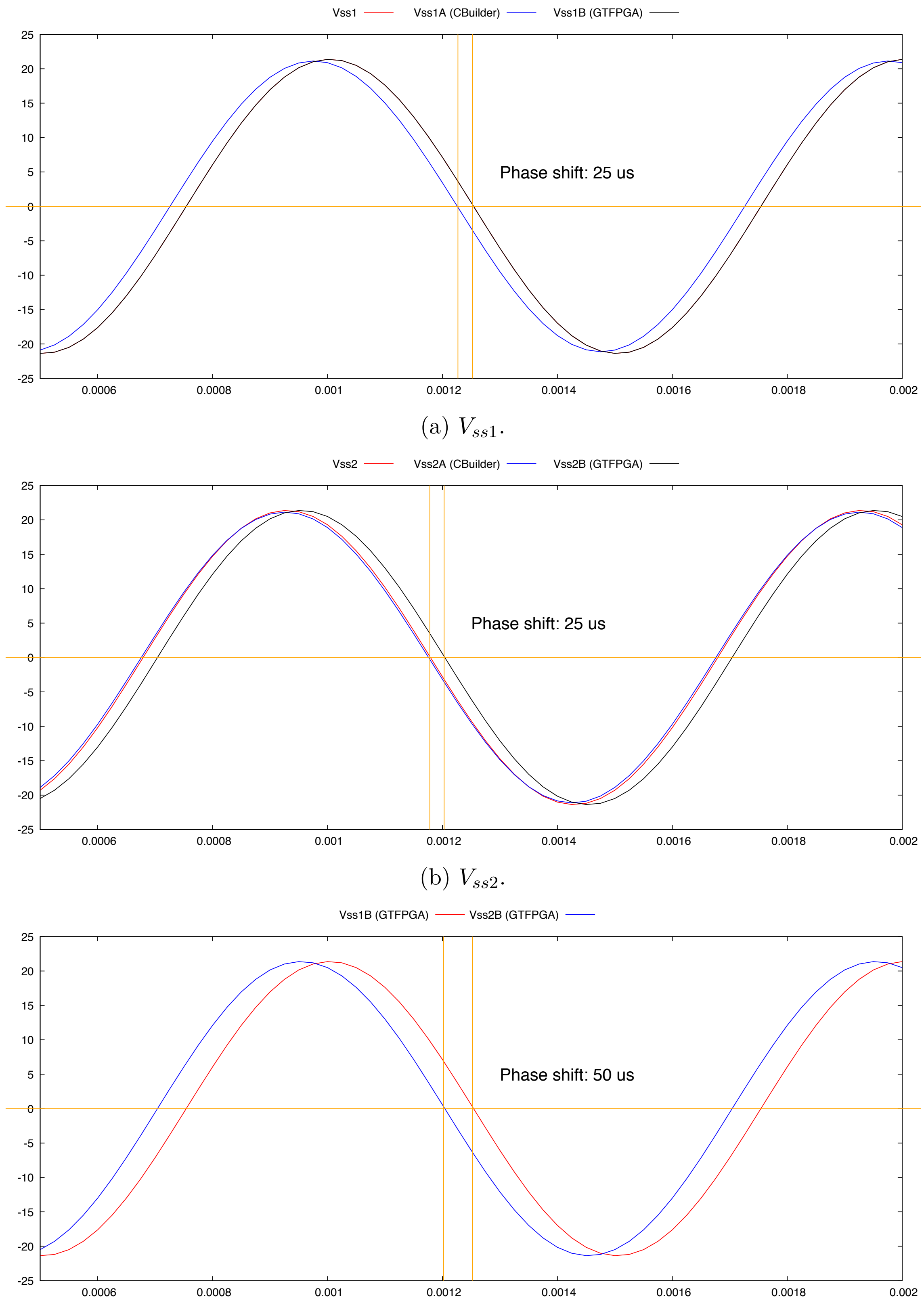
(a) Integrated version.

(b) Decoupled subsystems.

(c) Decoupled subsystem in CBuilder component.

(d) Decoupled subsystem 2 in Linux CPU via GTFPGA.

Figure 5.7: RSCAD drafts of simple circuit.



(a) V_{ss1}.

(b) V_{ss2}.

(c) V_{ss1b} & V_{ss2b}.

Figure 5.8: Interface quantities of simple circuit model with a fundamental frequency $f_0 = 1kHz$ and time step of 25 µs.

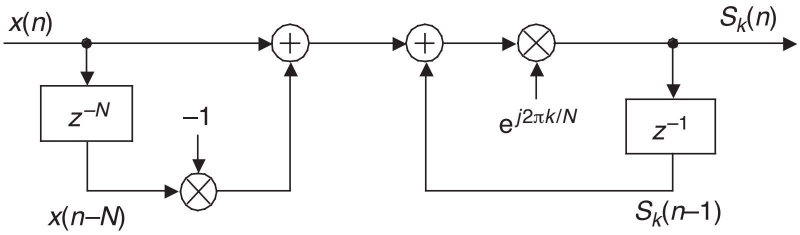


Figure 5.9: Sliding DFT algorithm [12].

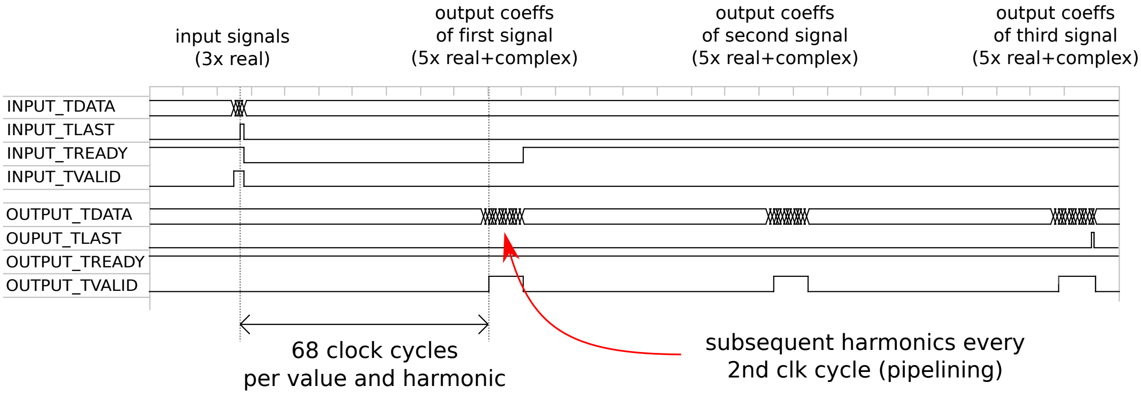


Figure 5.10: Wave dump of `hls_dft`.

Using Vivado's integrated logic analyzer and debug features, the performance of the implementation has been evaluated. The calculation of a single harmonic per value and time step requires **68 clock cycles** (544 ns). Thanks to a pipelined design the calculation of subsequent harmonics only takes **2 additional clock cycles** (16 ns) as shown in the wave dump 5.10. For three values per time step and five harmonics this sums up to 110 clock cycles (880 ns).

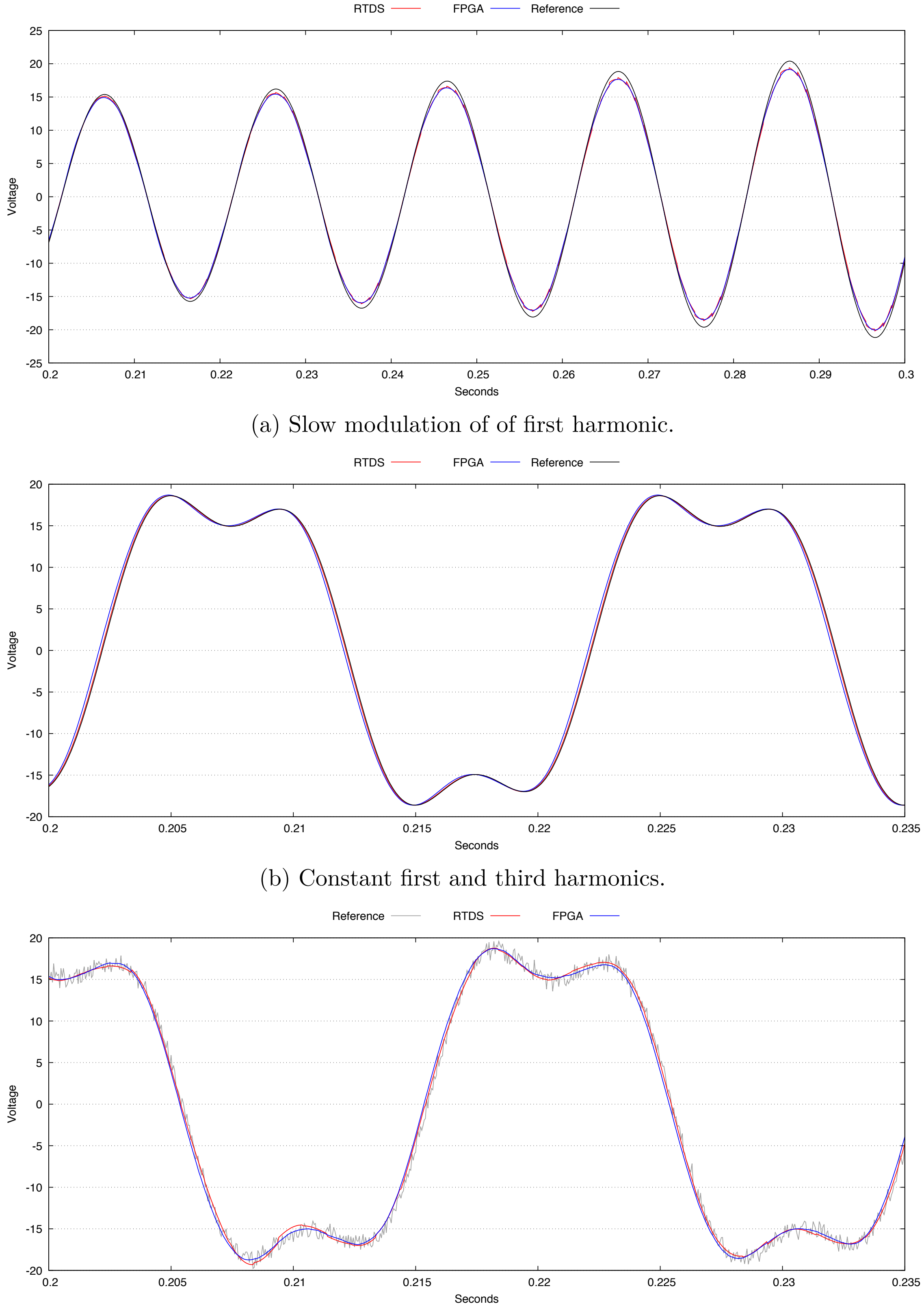
As usual, the IP core is controlable via an AXI4-Lite interface. This allows for adjusting the number and the frequency of the calculated harmonics. Optionally, a decimation ratio can be applied to reduce the amount of samples which are processed by the DP-model on the CPU.

Loopback Test

For testing, RTDS sends time-domain data to the previously described DFT implementation on the FPGA. After computation, the FPGA sends back the complex coefficients to RTDS. The signal is then reconstructed and compared with the reference signal as well as the reconstructed signal which has been transformed with RTDS's integrated `DFT32` block.

The generation of a reference signal and its reconstruction from the complex DFT coefficients is handled on the simulator itself. For the reference signal the sum of up to five sinusoids with varying frequency, magnitude and phase is used. This signal is then sampled and delayed based on user parameters. Both DFTs, the integrated and the external one on the FPGA, calculate the DC, 1st, 3rd, 5th and 7th harmonic components of the reference signal.

The plots in figure 5.11 display the results of the loopback test. It shows that the DFT on the FPGA can achieve better results than the integrated DFT from RTDS. The inherent latency of a single time step is almost imperceptible as the signal is reconstructed with reference to the current simulation time directly in RTDS.

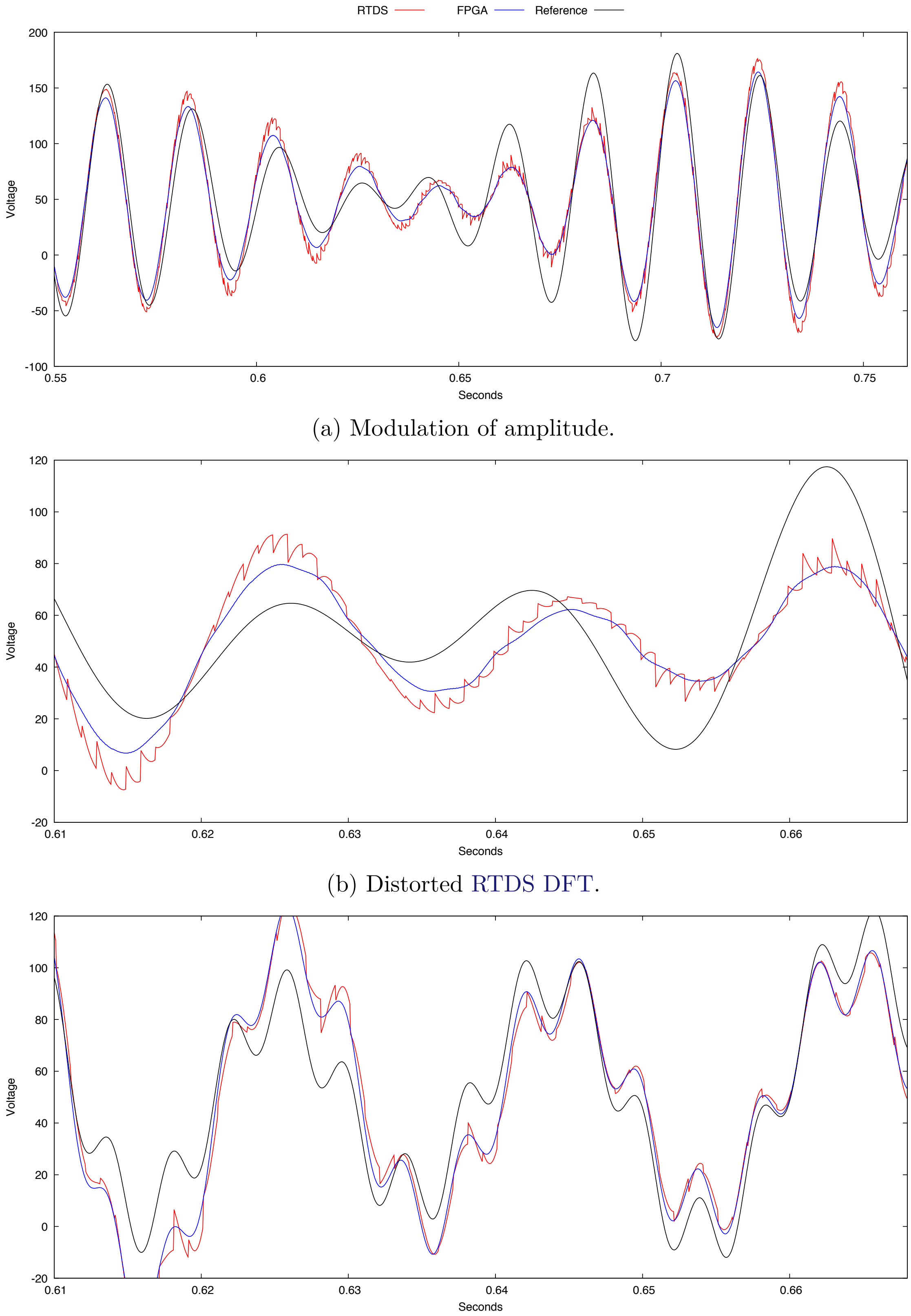


(a) Slow modulation of of first harmonic.

(b) Constant first and third harmonics.

(c) Noisy first and third harmonics.

Figure 5.11: RTDS versus HLS DFT implementations for slow transients.



(a) Modulation of amplitude.

(b) Distorted RTDS DFT.

(c) More harmonics and slow modulation.

Figure 5.12: RTDS versus HLS DFT implementations for fast transients.

6 Conclusion

This work shows that real-time co-simulation of power systems in a heterogenous environment is feasible. The GTFPGA netlist has been used to build a synchronized interface between RTDS and a standard Linux application. Time steps as low as 25 µs can be synchronized while only introducing a single time step latency per direction. This latency can be decreased even further if a serial communication pattern is used. In this case the round-trip time is equal to a single time step if the round is started by RTDS. These numbers align with the results gathered by first tests of the interface between RTDS and OPAL-RT which have been conducted previously.

The Xilinx Vivado IP integrator is well suited to build complex co-simulation setups consisting of more than two targets. Its support for industry standard AXI4 interfaces enables the reuse of existing AXI4 infrastructure IP cores like data-movers and the easy integration with modelling tools like the Vivado System Generator or Vivado High-level Synthesis. These tools allow the implementation of new models on the FPGA without deeper knowledge of a hardware description language like VHDL. However, this flexibility comes at the cost of a complex design flow as described in chapter 4.6. Therefore, a certain degree of FPGA design related knowledge is required to get started.

The integration with existing *VILLASnode* software enables a wide range of applications and interfaces to existing web services or databases like FIWARE. The execution of models on the Linux host machine is possible for time step above 25 µs. This is usually sufficient for EMT-based co-simulation which is mostly using a 50 µs time step period. Special optimizations are required for models which are executed below this period and interfaced with the FPGA. As these optimizations require rigorous measures they are not recommended. Instead, it is often easier to port the model to the FPGA itself. Thereby, the communication overhead between the FPGA and Linux can be eliminated. In comparison, commercial products like OPAL-RT's Linux-based eMegaSim simulator are limited to a minimum time step of 20 µs.

6.1 Future work

The product of this thesis is a versatile and extensible framework for the real-time co-simulation of power systems and interfaces to simulators for other purposes such as controller-in-the-loop. Two simple examples have been showcased to demonstrate the functionality and performance of *VILLASfpga*. These designs should act as a foundation for future applications like the ones which have been mentioned in section 1.2.1. Others include the addition of new interfaces to common industry field busses like EtherCAT or PowerLink. PHIL amplifiers feature their own interfaces like OPAL-RT's ORION or several custom designed links based on Xilinx's Aurora protocol [5]. Xilinx's new HLS or the System Generator tools enable the implementation of FPGA-based models by researchers who never worked with FPGAs before. The new interface to the RTDS simulator offers a large range of new possibilities like for example new flexible control algorithms which can be implemented in Linux.

Apart from new applications, several details of the framework can be improved. In theory, some GTFPGA blocks are supported in RTDS' small time step mode (VSC). This mode can be used to reduce the cycle time between FPGA and RTDS even below 10 µs. There is a chance that this mode can be used to overcome the maximum number of 64 values which can be exchanged per time step. GTFPGA blocks for the VSC mode already hint the possibility to use time division multiplexing to transfer a large set of values by distributing them over multiple time steps. Other ways to overcome this restriction are by adding more SFP modules to the FPGA board by using an extension module as shown in section 4.1.1. Also, reverse engineering of the GTFPGA code can be an option. RTDS is shipping PowerPC assembly `.mc` files for its components with the RSCAD suite. Disassembly tools like Radare[1] give insights into the internal program flow of these blocks. The overall structure and function calls of them are similar to custom CBuilder components. This suggests that there could be a chance to build a custom version of the GTFPGA block.

There are also ways to improve communication latency and determinism of the PCIe interface. In the current design, DMA controllers are used for data transfer between the FPGA-board and system memory. A paper by Flajslik et al. demonstrates a technique which reduces the latency by using the Last-level cache (LLC) to bypass the system memory [9]. However, this form of PIO requires deeper changes in the systems' memory management which are feasible with VFIO. They also introduce a concept which they call Network Interface Quibbles (NIQ) polling which artificially delays the completion of a read request to interrupt status registers. By doing so, they can effectively halt the CPU core for up to 50 ms[2]. If 50 ms have passed and still no interrupt has occurred, the CPU will issue a new read request to the same interrupt status register. In case of an event, the PCIe card will promptly complete the read request with an appropriate response.

Section 4.4.1 describes limitations when executing hard real-time tasks on Linux. More optimizations and special hardware can improve this. Specialized Real-time Operating Systems (RTOSs) like VxWorks might be the best way to solve this issue. Wassen and Lankes showed how modern Non-uniform Memory Access (NUMA) systems can improve the real-time performance by isolating complete sockets of the system [29]. On such more strictly isolated CPU cores, baremetal applications or Asymmetric Multiprocessing (AMP) with a unikernel like HermitCore could be tested [15]. Alternatively, a completely different architecture like Xilinx's new Zynq-

[1] http://radare.org

[2] This is the limit imposed by the PCIe specification

based systems could be used. Section 3.4 describes the advantages of a Zynq-based platform because of their tighter integration of PL and ARM-based CPU cores in a single chip.

A more theoretical problem is the search of suitable communication patterns for co-simulation scenarios between more than two simulators. *VILLASfpga* is prepared to support data exchange in such setups. Yet, the synchronization of such complex topologies might contain an open question.

Automation of the framework utilization is a never-ending task. The integration of *VILLASnode* and *VILLASfpga* can be further simplified. Currently, the user has to provide a configuration file which lists all available models, interfaces and auxiliary IP cores. This file can be automatically generated by using various side output products of the Xilinx design flow as described in chapter 4.6.

Part I

Appendix

A Code examples

A.1 *VILLASnode* Configuration File for *VILLASfpga*

```
# Example configuration file for VILLASfpga / VILLASnode
#
# The syntax of this file is similar to JSON.
# A detailed description of the format can be found here:
#  http://www.hyperrealm.com/libconfig/libconfig_manual.html
#
# Author:  Steffen Vogel <stvogel@eonerc.rwth-aachen.de>
# Copyright:  2016, Steffen Vogel
##

############       Global Options      ############

affinity = 0x0C;    # Mask of cores the server should run on
                    # This also maps the NIC interrupts to
                       ↪ those cores!

priority = 50;      # Priority for the server tasks.
                    # Usually the server is using a
                       ↪ real-time FIFO
                    # scheduling algorithm

############     FPGA configuration    ############

fpga = {
  id = "10ee:7022"; # Card identification
  slot = "01:00.0"; # Usually only id or slot is required

  do_reset = true;  # Perform a full reset of the FPGA board
                    # Requires a IP core named 'axi_reset_0'

  ############   List of IP cores on FPGA   ############
  #
  # Every IP core can have the following settings:
  #   baseaddr  Baseaddress as accessible from BAR0 memory
     ↪ region
```

```
#   irq        Interrupt index of MSI interrupt controller
#   port       Port index of AXI4-Stream interconnect

ips = {
  ### Utility IPs
  axi_pcie_intc_0 = {
    vlnv       =
       ↪ "acs.eonerc.rwth-aachen.de:user:axi_pcie_intc:1.0";
    baseaddr   = 0xb000;
  },
  switch_0 = {
    vlnv       = "xilinx.com:ip:axis_interconnect:2.1"
    baseaddr   = 0x5000;
    num_ports  = 10;
  },
  axi_reset_0 = {
    vlnv       = "xilinx.com:ip:axi_gpio:2.0";
    baseaddr   = 0x7000;
  },
  timer_0 = {
    vlnv       = "xilinx.com:ip:axi_timer:2.0";
    baseaddr   = 0x4000;
    irq        = 0;
  },

  ### Data mover IPs
  dma_0 = {
    vlnv       = "xilinx.com:ip:axi_dma:7.1";
    baseaddr   = 0x3000;
    port       = 1;
    irq        = 3;      # 3 = MM2S, 4 = S2MM
  },
  dma_1 = {
    vlnv       = "xilinx.com:ip:axi_dma:7.1";
    baseaddr   = 0x2000;
    port       = 6;
    irq        = 3;      # 3 = MM2S, 4 = S2MM
  },
  fifo_mm_s_0 = {
    vlnv = "xilinx.com:ip:axi_fifo_mm_s:4.1";
    baseaddr   = 0x6000;
    baseaddr_axi4 = 0xC000;
    port       = 2;
    irq        = 2;      # MM2S + S2MM
  },
    ### Interface IPs
    rtds_axis_0 = {
      vlnv      =
         ↪ "acs.eonerc.rwth-aachen.de:user:rtds_axis:1.0";
      baseaddr  = 0x8000;
      port      = 0;
      irq       = 5;     # 5 = TS, 6 = Overrun, 7 = Case
    },

    ### Model IPs
    hls_dft_0 = {
      vlnv      =
         ↪ "acs.eonerc.rwth-aachen.de:hls:hls_dft:1.0";
      baseaddr  = 0x9000;
      port      = 5;
      irq       = 1;

      period    = 400;    # In samples: 20ms / 50uS = 400
      harmonics = [ 0, 1, 3, 5, 7 ];
      decimation = 0;     # 0 = disabled */
    },
    axis_data_fifo_0 = {
      vlnv      = "xilinx.com:ip:axis_data_fifo:1.1";
      port      = 3;
    },
    axis_data_fifo_1 = {
      vlnv      = "xilinx.com:ip:axis_data_fifo:1.1";
      port      = 6;
    },
  }

  ############   Switch config   ############
  # Connect IP core in VILLASfpga
  # Requires a single IP core with VLNV:
  #  xilinx.com:ip:axis_interconnect
  # And correct 'port' settings per IP

  paths = (
// { in = "fifo_mm_s_0", out = "fifo_mm_s_0" },
// { in = "dma_0",       out = "dma_0" },
// { in = "dma_1",       out = "dma_1" }
// { in = "rtds_axis_0", out = "fifo_mm_s_0", reverse = true
   ↪ }
// { in = "rtds_axis_0", out = "dma_0",       reverse = true
   ↪ }
   { in = "rtds_axis_0", out = "dma_1",       reverse = true }
// { in = "rtds_axis_0", out = "fifo_mm_s_0", reverse = true }
// { in = "dma_0",       out = "hls_dft_0",   reverse = true }
// { in = "rtds_axis_0", out = "hls_dft_0",   reverse = true }
  )
}

############    List of plugins    ############
#
# Additional node-types, hooks or VILLASfpga IP cores
# can be loaded by compiling them into a shared library and
# adding them to this list

plugins = [
  "./lib/cbmodels/simple_circuit.so" # Providing 'cbuilder' model 'simple_circuit'
]

############    Dictionary of nodes    ############

nodes = {
  dma_0 = {
    type      = "fpga";     # Datamovers to VILLASfpga
    datamover = "dma_0";    # Name of IP core in fpga.ips
    use_irqs  = false;      # Use polling or MSI interrupts?
  },
  dma_1 = {
    type      = "fpga";
    datamover = "dma_1";
    use_irqs  = false;
  },
  fifo_0 = {
    type      = "fpga";
    datamover = "fifo_mm_s_0";
    use_irqs  = false;
  },
  simple_circuit = {
    type      = "cbuilder";
    model     = "simple_circuit",
    timestep  = 25e-6;
    parameters = [
      1.0,                  # R2 = 1 Ohm
      0.001                      # C2 = 1000 uF
    ];
  }
}

############       List of paths       ############

paths = (
  { in = "dma_1", out = "simple_circuit", reverse = true }
)
```

Listing A.1: Example configuration for *VILLASnode*

A.2 XML: Accelerator Map of hls_multiply

```
<?xml version="1.0" encoding="utf-8"?>
<xd:acceleratorMap xmlns:xd="http://www.xilinx.com/xidane"
   ↪ xd:functionName="hls_multiply"
   ↪ xd:componentRef="hls_multiply"
   ↪ xd:initiationInterval="x" xd:clockPeriod="8.000000"
   ↪ xd:sequential="true" xd:hostMachine="64"
   ↪ xd:reset="control">
  <xd:arg xd:name="input.data" xd:originalName="input"
     ↪ xd:direction="in" xd:interfaceRef="input_r"
     ↪ xd:busTypeRef="axis" xd:dataWidth="32"/>
  <xd:arg xd:name="input.last" xd:originalName="input"
     ↪ xd:direction="in" xd:interfaceRef="input_r"
     ↪ xd:busTypeRef="axis" xd:dataWidth="1"/>
  <xd:arg xd:name="output.data" xd:originalName="output"
     ↪ xd:direction="out" xd:interfaceRef="output_r"
     ↪ xd:busTypeRef="axis" xd:dataWidth="32"/>
  <xd:arg xd:name="output.last" xd:originalName="output"
     ↪ xd:direction="out" xd:interfaceRef="output_r"
     ↪ xd:busTypeRef="axis" xd:dataWidth="1"/>
  <xd:arg xd:name="factors[]" xd:originalName="factors"
     ↪ xd:direction="in" xd:interfaceRef="s_axi_config"
     ↪ xd:busTypeRef="axilite" xd:offset="0x80"
     ↪ xd:arraySize="32" xd:dataWidth="32"/>
  <xd:latencyEstimates xd:best-case="undef"
     ↪ xd:worst-case="undef" xd:average-case="undef"/>
  <xd:resourceEstimates xd:LUT="348" xd:FF="335" xd:BRAM="2"
     ↪ xd:DSP="3"/>
```

```
</xd:acceleratorMap>
```

Listing A.2: XML Accelerator Map of `hls_multiply`

A.3 VHDL: `hdl_multiply`

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

entity hdl_multiply is
  generic (
    FACTOR        : real := 3.3
  );
  port (
    clk       : in  std_logic;
    aresetn       : in  std_logic;
    s_axis_input_tdata    : in  std_logic_vector(31 downto 0);
    s_axis_input_tvalid   : in  std_logic;
    s_axis_input_tlast    : in  std_logic;
    s_axis_input_tready   : out std_logic;
    m_axis_output_tdata   : out std_logic_vector(31 downto 0);
    m_axis_output_tvalid    : out std_logic;
    m_axis_output_tlast   : out std_logic;
    m_axis_output_tready    : in  std_logic
  );
end entity;

architecture rtl of hdl_multiply is
  component multiply_floating_point_v7_1_0
  port (
    aclk        : in  std_logic;
    aclken        : in  std_logic;
    s_axis_a_tvalid     : in  std_logic;
    s_axis_a_tdata      : in  std_logic_vector(31 downto 0);
    s_axis_a_tlast      : in  std_logic;
    s_axis_a_tready     : out std_logic;
    s_axis_b_tvalid     : in  std_logic;
    s_axis_b_tdata      : in  std_logic_vector(31 downto 0);
    s_axis_b_tlast      : in  std_logic;
    s_axis_b_tready     : out std_logic;
    m_axis_result_tvalid    : out std_logic;
    m_axis_result_tdata   : out std_logic_vector(31 downto 0);
    m_axis_result_tlast   : out std_logic;
    m_axis_result_tready    : in  std_logic
  );
  end component;
begin

-- IP generated with Vivado
MULTP : multiply_floating_point_v7_1_0
  port map (
    aclk        => clk,
    aclken        => '1',
    -- Input
    s_axis_a_tvalid     => s_axis_input_tvalid,
    s_axis_a_tdata      => s_axis_input_tdata,
    s_axis_a_tlast      => s_axis_input_tlast,
    s_axis_a_tready     => s_axis_input_tready,
    -- Factor
    s_axis_b_tvalid     => '1',
    s_axis_b_tdata      => to_slv(to_float(FACTOR)),
    s_axis_b_tlast      => '0',
    s_axis_b_tready     => open,
    -- Result
    m_axis_result_tvalid    => m_axis_output_tvalid,
    m_axis_result_tdata   => m_axis_output_tdata,
    m_axis_result_tlast   => m_axis_output_tlast,
    m_axis_result_tready    => m_axis_output_tready
  );

end architecture;
```

Listing A.3: VHDL code of `hdl_multiply`

A.4 HLS: `hls_dft`

```
#include <iostream>
#include <complex>

#include <hls_math.h>
#include <ap_shift_reg.h>

```

```
struct axis {
  float data;
  ap_uint<1> last;
};

/** DFT window period (1 / fundamental-frequency) */
const float PERIOD = 1.0 / 50; // in sec

/** Simulation time step (1 / sample-rate) */
const float TIMESTEP = 50e-6; // in sec

/** Number of samples in a window */
const int NSAMPLES = 400;//PERIOD / TIMESTEP;

/** Number of values per sample */
const int MAX_VALUES = 8;

/** Number of harmonics */
const int MAX_HARMONICS = 16;

/* Single precision pi constant becuase M_PI is double! */
const float pi = 3.141592653589793238462643383279502884f;

void hls_dft(stream<axis> &input, stream<axis> &output,
    ↪ float fharmonics[MAX_HARMONICS], ap_int<8>
    ↪ num_harmonics, ap_int<8> decimation) {
  #pragma HLS INTERFACE s_axilite
      ↪ port=return,fharmonics,num_harmonics,decimation
      ↪ bundle=ctrl
  #pragma HLS INTERFACE axis port=input,output
  #pragma HLS STREAM depth=64 variable=input,output
  #pragma HLS DATAFLOW

  /** Previous coefficients for incremental update */
  static complex<float> coeffs[MAX_HARMONICS];

  /* Time */
  static float t;
  static ap_int<32> decimation_cnt;

  /** Sliding window of samples */
  static ap_shift_reg<float,NSAMPLES> windows[MAX_VALUES];

  /** AXI Stream signals */
  axis real, imag, refph;

```

```
LOOP_VALUES:
  for (int index = 0; index < MAX_VALUES; index++) {
    /* Read real-valued time-domain data from AXI Stream interface */
    axis in = input.read();

    /* Shift and get data from SLR */
    float newest = in.data;
    float oldest = windows[index].shift(newest, NSAMPLES-1);

LOOP_HARMONICS:
    for (int i = 0; i < num_harmonics; i++) {
      #pragma HLS PIPELINE II=2

      float pi_fharm = 2.0f * pi * fharmonics[i];

      /* Recursive update */
      coeffs[i] = polar<float>(1.0f, pi_fharm) * (coeffs[i] + (newest - oldest));

      /* Correction for stationary phasor */
      complex<float> correction = polar<float>(1.0f, pi_fharm * (t - (NSAMPLES + 1)));
      complex<float> result = 2.0f / NSAMPLES * coeffs[i] / correction;

      /* DC component */
      if (i == 0)
        result /= 2.0f;

      /* Update real part */
      real.data = result.real();
      real.last = 0;

      /* Update imaginary part */
      imag.data = result.imag();
      imag.last = 0;

      /* Only every'th decimation_cnt'th sample will be sent via AXI4-Stream */
      if (decimation_cnt == 0) {
        output.write(real);
        output.write(imag);
      }
    }

    if (in.last)
      break; /* start next packet */
  }

  /* The decimation_cnt suppresses the output of results */
  if (decimation_cnt == 0) {
    decimation_cnt = decimation;

    /* Last word on AXI-Stream bus is the reference phase */
    refph.data = t++;
    refph.last = 1;
    output.write(refph);
  }
  else
    decimation_cnt--;
}
```

Listing A.4: HLS implementation of recursive DFT

A.5 Linux configuration

```
cmdline = intel_iommu=on isolcpus=2,3 nohz=on nohz_full=2,3
   ↪ nosoftlockup intel_pstate=disable
```

Listing A.5: Linux boot command line.

```
CONFIG_PREEMPT_RT_FULL=y
CONFIG_DMAR_DEFAULT_ON=y
CONFIG_VFIO_IOMMU_TYPE1=m
CONFIG_VFIO_VIRQFD=m
CONFIG_VFIO=m
CONFIG_VFIO_NOIOMMU=y
CONFIG_VFIO_PCI=m
CONFIG_VFIO_PCI_MMAP=y
CONFIG_VFIO_PCI_INTX=y
```

Listing A.6: Extract of Linux kernel build-time configuration.

B List of Acronyms

API	Application Programming Interface
GUI	Graphical User Interface
IDE	Integrated Development Environment
IO	Input / Output
LCD	Liquid-crystal Display
OS	Operating System
PC	Personal Computer
PWM	Pulse-width Modulation
SMA	Sub-Miniature-A
SW	Software
TSDB	Time-series Database
VM	Virtual Machine
	Institutions
CAPS	Center for Advanced Power Systems
ERIC-LAB	European Real-time Integrated Co-simulation laboratory
IBM	International Business Machines Corporation
	Protocols / Interfaces / Busses
AHCI	Advanced Host Controller Interface
ATA	AT Attachment
COMTRADE	Common format for Transient Data Exchange for power systems
NTP	Network Time Protocol
ORION	Opal Remote IO Network
SCI	Scalable Coherent Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USB	Universal Serial Bus

XML	Extensible Markup Language

RTDS

3PC	Tripple Processor card
GBH	Global Bus Hub
GPC	Giga Processor card
GTFPGA	Giga Tranceiver Field Programmable Gate Array
GTNET	Giga Tranceiver Network card
GTSYNC	Giga Tranceiver Synchronization card
GTWIF	Giga Tranceiver Workstation Interface card
GT	Giga Tranceiver
IRC	Inter Rack Communication switch
PB5	PB5 Processor card
RPC	RISC Processor card
RSCAD	RTDS Simulator Software
RTDS	Real-time Digital Simulator
TPC	Tandem Processor card

Simulation

DDS	Distributed Data Structures
DRTS	Digital Real-Time Simulation
HIL	Hardware-in-the-Loop
HLA	High Level Architecture
PHIL	Power-Hardware-in-the-Loop
RCP	Rapid Control Prototyping
RTT	Round-trip Time
TAI	Temps Atomique International
GPS	Global Positioning System
GLONASS	Globalnaja Nawigazionnaja Sputnikowaja Sistema
BMC	Best Master Clock
PTP	Precision Time Protocol (IEEE 1588)
IRIG-B	Inter Range Instrumentation Group Timecode B
PPS	Pulse per Second

Power Systems

EMT	Electro-magnetic transient
DP	Dynamic Phasor
DFT	Discrete Fourier Transform
AES	All-Electric Ship
VSC	Voltage Source Converter
MMC	Modular Multi-level Converter
ITM	Ideal Transformer Model
PMU	Phasor Measurement Unit
SCADA	Supervisory Control and Data Acquisition
DNP3	Distributed Network Protocol
SV	Sampled Values (IEC 61850-9-2)
GOOSE	Generic Object Oriented Substation Events (IEC 61850-8-1)

Computer Architecture

AMP	Asymmetric Multiprocessing
KVM	Kernel-based Virtual Machine
PF	Page Fault
HPC	High-performance Computing
SoC	System-on-Chip
NoC	Network-on-Chip
LLC	Last-level cache
AMBA	Advanced Microcontroller Bus Architecture
AXI	Advanced eXtensible Interface Bus
AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
PLB	Processor Local Bus
VFIO	Virtual Function IO
UIO	Userspace IO
PCI	Peripheral Component Interconnect
PCIe	PCI Express
TLP	PCIe Transaction Layer Packet

ACS	PCIe Access Control Services
EP	PCIe Endpoint
BAR	PCIe Base Address Register
MSI	PCIe Message Signalled Interrupt
DMA	Direct Memory Access
ECC	Error-Correcting Code
MMIO	Memory-mapped IO
NUMA	Non-uniform Memory Access
PIO	Programmed IO
RAM	Random-Access Memory
ROM	Read-Only Memory
RMA	Remote Memory Access
SG	Scatter-Gather
VA	Virtual Address
WC	Write-combine
CPU	Central Processing Unit
GPU	Graphic Processing Unit
MMU	Memory Management Unit
IOMMU	Input / Output MMU
DSP	Digital Signal Processor
PIC	Programmable Interrupt Controller
NIC	Network Interface Controllers
ISR	Interrupt Service Routine
IRQ	Interrupt Request
SMI	System Management Interrupt
NMI	Non-maskable Interrupt
NIQ	Network Interface Quibbles

FPGA Design

ASIC	Application-specific Integrated Circuit
CDC	Clock domain crossing
ESL	Electronic System-level Design
FIFO	First-in First-out

FMC	FPGA Mezzanine Card
FPGA	Field Programmable Gate Array
GTX	Gigabit Tranceiver
HDL	Hardware Description Language
IP	Intellectual Property
LC	Lucent Connector
MM2S	Memory-Map-to-Stream
PL	Programmable Logic
PROM	Programmable Read-only Memory
RTL	Register Transfer Level
S2MM	Stream-to-Memory-Map
SFP	Small Form-factor Pluggable
SGMII	Serial Gigabit Media-independent Interface
VHDL	VHSIC Hardware Description Language
ISE	Xilinx Integrated Synthesis Environment
EDK	Xilinx Embedded Development Kit
IPI	Vivado IP Integrator
HLS	Vivado High-level Synthesis
XSG	Vivado System Generator for DSP

List of Figures

List of Tables

List of Listings

Bibliography

[1] ARM. *AMBA AXI and ACE Protocol Specification*. 4th ed. Feb. 2013.

[2] Bernhard Baumgartner, Christian Riesch, and Manfred Rudigier. "IEEE 1588/PTP: The Future of Time Synchronization in the Electric Power Industry". In: *PAC World Conference*. 2012. URL: https://www.omicron-lab.com/fileadmin/assets/customer_examples/IEEE_1588_PTP_-_The_Future_of_Time_Synchronization_in_the_Electric_Power_Industry.pdf (visited on 07/02/2016).

[3] J. Belanger, P. Venne, and J. N. Paquin. "The what, where and why of real-time simulation". In: *Planet RT* 1 (2010), p. 1.

[4] A. Benigni et al. "FlePS: A power interface for Power Hardware In the Loop". In: *Proceedings of the 2011-14th European Conference on Power Electronics and Applications (EPE 2011)*. Proceedings of the 2011-14th European Conference on Power Electronics and Applications (EPE 2011). Aug. 2011, pp. 1–10.

[5] Andreas Berger. "Expanding data exchange capabilities for RTDS using an industry based communication protocol". Center for Advanced Power Systems (CAPS), Florida State University, Sept. 2013. URL: https://www.akademikerverlag.de/catalog/details//store/de/book/978-3-639-80778-3/expanding-data-exchange-capabilities-for-rtds.

[6] Carl Bisaillon and Gernot Pammer. "Power Hardware-In-the-Loop (PHIL): Optic Fiber ORION Protocol". The 7th International Conference on Real-Time Simulation Technologies. Montréal, June 2014. URL: http://www.opal-rt.com/sites/default/files/RT14_EGSTON_PHIL.pdf.

[7] Jean-François Cécile et al. "A Distributed Real-Time Framework For Dynamic Management Of Heterogeneous Co-simulations". In: *Opal-RT Technologies Inc., Montréal, Canada* (2006). URL: http://www.rtlabcentral.com/sites/default/files/technical_papers/scs_article.pdf (visited on 06/04/2016).

[8] M. D. Omar Faruque et al. "Real-Time Simulation Technologies for Power Systems Design, Testing, and Analysis". In: *IEEE Power and Energy Technology Systems Journal* 2.2 (June 2015), pp. 63–73. ISSN: 2332-7707. DOI: 10.1109/JPETS.2015.2427370.

[9] Mario Flajslik and Mendel Rosenblum. “Network Interface Design for Low Latency Request-response Protocols”. In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC’13. Berkeley, CA, USA: USENIX Association, 2013, pp. 333–346. URL: http://dl.acm.org/citation.cfm?id=2535461.2535502 (visited on 05/03/2016).

[10] Yizhong Hu et al. “Development of an RTDS-TSA hybrid transient simulation platform with frequency dependent network equivalents”. In: *IEEE PES ISGT Europe 2013*. IEEE PES ISGT Europe 2013. Oct. 2013, pp. 1–5. DOI: 10.1109/ISGTEurope.2013.6695283.

[11] *Input–output memory management unit*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 723097455. June 1, 2016. URL: https://en.wikipedia.org/w/index.php?title=Input%C3%A2%C2%80%C2%93output_memory_management_unit&oldid=723097455 (visited on 06/16/2016).

[12] E. Jacobsen and R. Lyons. “An update to the sliding DFT”. In: *IEEE Signal Processing Magazine* 21.1 (Jan. 2004), pp. 110–111. ISSN: 1053-5888. DOI: 10.1109/MSP.2004.1516381.

[13] E. Jacobsen and R. Lyons. “The sliding DFT”. In: *IEEE Signal Processing Magazine* 20.2 (Mar. 2003), pp. 74–80. ISSN: 1053-5888. DOI: 10.1109/MSP.2003.1184347.

[14] R. Kuffel et al. “RTDS-a fully digital power system simulator operating in real time”. In: *, IEEE WESCANEX 95. Communications, Power, and Computing. Conference Proceedings.* , IEEE WESCANEX 95. Communications, Power, and Computing. Conference Proceedings. Vol. 2. May 1995, 300–305 vol.2. DOI: 10.1109/WESCAN.1995.494045.

[15] Stefan Lankes, Simon Pickartz, and Jens Breitbart. “HermitCore: A Unikernel for Extreme Scale Computing”. In: ACM Press, 2016, pp. 1–8. ISBN: 978-1-4503-4387-9. DOI: 10.1145/2931088.2931093. URL: http://dl.acm.org/citation.cfm?doid=2931088.2931093 (visited on 05/26/2016).

[16] OPAL-RT Technologies. *E.ON Energy Research Center builds first interface between OPAL-RT and RTDS Technologies real-time simulators, opens a new era of collaborative research opportunities*. Feb. 2016. URL: http://www.opal-rt.com/press-release/eon-energy-research-center-builds-first-interface-between-opal-rt-and-rtds-technologie.

[17] OPAL-RT Technologies. *RT-LAB Distributed Real-Time Power - Product Brochure*. 2016. URL: http://www.opal-rt.com/sites/default/files/rtlab_brochure_en.pdf.

[18] Jean-Nicolas Paquin et al. “A modern and open real-time digital simulator of All-Electric Ships with a multi-platform co-simulation approach”. In: *Electric Ship Technologies Symposium, 2009. ESTS 2009. IEEE*. IEEE, 2009, pp. 28–35. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4906490 (visited on 06/04/2016).

[19] *PCI Express*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 725261885. June 14, 2016. URL: https://en.wikipedia.org/w/index.php?title=PCI_Express&oldid=725261885 (visited on 06/16/2016).

[20] *Product Guide: AXI Interconnect v2.1*. Apr. 2016. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf (visited on 06/16/2016).

[21] Kavya Rentachintala. "Multi-Rate Co-Simulation Interfaces Between The RTDS And The Opal-RT". Florida State University, Apr. 2012. URL: http://diginole.lib.fsu.edu/etd/5134.

[22] RTDS Technologies. *Florida State University pushes for development of GTFPGA*. Summer 2009. URL: https://www.rtds.com/wp-content/uploads/2014/11/RTDSNEWS_2009_summer.pdf (visited on 05/06/2016).

[23] RTDS Technologies. *GTSYNC Synchronization Card*. May 2013.

[24] Michael Sloderbeck et al. "High-speed Digital Interface for a Real-time Digital Simulator". In: *Proceedings of the 2010 Conference on Grand Challenges in Modeling & Simulation*. GCMS '10. Ottawa, Ontario, Canada: Society for Modeling & Simulation International, 2010, pp. 399–405. URL: http://dl.acm.org/citation.cfm?id=2020619.2020674.

[25] M. J. Stanovich et al. "Multi-agent testbed for emerging power systems". In: *2013 IEEE Power Energy Society General Meeting*. 2013 IEEE Power Energy Society General Meeting. July 2013, pp. 1–5. DOI: 10.1109/PESMG.2013.6672944.

[26] Mark Stanovich, Mike Sloderbeck, and Raveendra Meka. "Connecting the RTDS to a Multi-Agent System Testbed Utilizing the GTFPGA". RTDS User's Group Meeting. Tallahassee, Florida, 2013.

[27] *The American Heritage Dictionary of the English Language*. Boston: Houghton Mifflin Company, 2000.

[28] Liang-min Wang. *How to Implement a 64B PCIe* Burst Transfer on Intel® Architecture*. Feb. 2013. URL: http://www.intel.de/content/dam/www/public/us/en/documents/white-papers/pcie-burst-transfer-paper.pdf (visited on 05/09/2016).

[29] Georg Wassen and Stefan Lankes. "Bare-Metal Execution of Hard Real-Time Tasks Within a General-Purpose Operating System". In: *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*. Ed. by Francisco J. Cazorla. Vol. 47. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 75–84. ISBN: 978-3-939897-95-8. DOI: http://dx.doi.org/10.4230/OASIcs.WCET.2015.75. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5258.

[30] Xilinx. *7 Series FPGAs Overview*. May 2015. URL: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[31] Xilinx. *Product Guide: AXI DMA v7.1*. Nov. 2015. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf (visited on 06/16/2016).

[32] Xilinx. *Product Guide: AXI GPIO v2.0*. Nov. 2015. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf (visited on 06/16/2016).

[33] Xilinx. *Product Guide: AXI Interrupt Controller (INTC) v4.1*. Apr. 2016. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_intc/v4_1/pg099-axi-intc.pdf (visited on 06/16/2016).

[34] Xilinx. *Product Guide: AXI Memory Mapped to PCI Express (PCIe) Gen2 v2.6*. June 2015. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_pcie/v2_6/pg055-axi-bridge-pcie.pdf (visited on 06/16/2016).

[35] Xilinx. *Product Guide: AXI Timer v2.0*. Nov. 2015. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf (visited on 06/16/2016).

[36] Xilinx. *Product Guide: AXI4-Stream FIFO v4.1*. Apr. 2016. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo-mm-s.pdf (visited on 06/16/2016).

[37] Xilinx. *Product Guide: AXI4-Stream Interconnect v1.1*. Nov. 2015. URL: http://www.xilinx.com/support/documentation/ip_documentation/axis_interconnect/v1_1/pg035_axis_interconnect.pdf.

[38] Xilinx. *Product Guide: FIFO Generator v12.0*. June 2015. URL: http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v12_0/pg057-fifo-generator.pdf (visited on 06/16/2016).

[39] Xilinx. *VC707 Evaluation Board for the Virtex-7 FPGA*. Mar. 2016. URL: http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf.