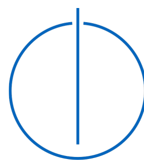
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Robotics, Cognition, Intelligence

Development of a Runtime Switch Behavior for a Multi-level RISC-V Instruction Set to Register Transfer Fault Injection Simulator

Lasse Urban



DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Robotics, Cognition, Intelligence

Development of a Runtime Switch Behavior for a Multi-level RISC-V Instruction Set to Register Transfer Fault Injection Simulator

Entwicklung eines Laufzeittransitionsverhaltens für einen multi-level RISC-V-Instruktionssatz- zu Registertransfer-Fehlerinjektionssimulator

Author:	Lasse Urban
Supervisor:	Prof. Dr.-Ing. habil. Daniel Müller-Gritschneder
Advisor:	Johannes Geier, M.Sc.
Supervising Chair:	Chair of Electronic Design Automation
	TUM Department of Electrical and Computer Engineering
Submission Date:	15. October 2022

Abstract

Systems on Chips (SoCs) have become indispensable in today's world and the demand for them is constantly increasing. For this reason, the need for efficient and secure methods to verify the functionality and safety of the chips is also growing.
An interesting and important approach for this is the fault injection simulation with virtual prototypes of hardware components and systems. For example, a virtual prototype of a central processing unit (CPU) can allow the simulation of soft error injection scenarios. These errors can be injected either at instructional level or at micro-architectural level. For this purpose, either a very performant instruction set simulator (ISS) or a slower register transfer level (RTL) core model can be used to implement the CPU of a virtual prototype. In this work, a runtime switch behavior for fault injection simulators based on virtual prototypes is presented. It can be used to execute a test program very fast on instruction level at the beginning. At a certain point in time when more detailed investigations are to be carried out, e.g., a fault in the micro-architecture, it can be switched to a CPU on RTL. Later, it is possible to switch back to the ISS level. The whole procedure can accelerate the simulation by a multiple compared to the simulation running solely on RTL. Using the Extendable Translating Instruction Set Simulator (ETISS) and an verilated RTL (VRTL) model of a RISC-V core, RI5CY, a factor of four was observed. The runtime switch behavior was implemented in the form of a CPU multiplexer that can switch between different CPU levels of abstraction at runtime. It also makes it possible to have several CPUs execute the same test program approximately synchronously at the same time.

Contents

Abstract **iii**

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Related Work . . . 4

2. Background Information **6**
2.1. Virtual Prototyping . . . 6
2.2. SystemC . . . 8
2.2.1. SystemC Simulation Kernel and Core Language Concepts . . . 8
2.2.2. Transaction Level Modeling (TLM-2.0) . . . 10
2.3. SystemVerilog . . . 13
2.4. Verilator . . . 14
2.5. Extendable Translating Instruction Set Simulator (ETISS) . . . 15
2.5.1. Architecture of ETISS . . . 15
2.5.2. Plugins for ETISS . . . 16
2.5.3. Virtual Core Structure in ETISS . . . 16
2.6. RI5CY RTL Core . . . 17
2.6.1. RISC-V Instruction Set Architecture Overview . . . 17
2.6.2. RI5CY Implementation . . . 18
2.7. Fault Injection Virtual Platform (FIVP) . . . 19

3. Conceptional Development of a Runtime Switch Behavior for Virtual Platforms **21**
3.1. Definition of a Runtime Switch Behavior . . . 21
3.2. Implementation Concept . . . 23
3.3. Development Flow . . . 26

4. Software Model of a CPU Multiplexer **28**
4.1. Clock and Reset Signal Forwarding . . . 28
4.2. Freezing and Waking up CPUs . . . 28
4.3. TLM Routing for Memory Reads and Writes . . . 29
4.4. Refinement Map . . . 32
4.5. Interrupt Handling . . . 33
4.6. Switch Functionality . . . 35

5. Experimental Evaluation - Results **37**
5.1. Evaluation Approach 37
5.2. Results 39

6. Summary and Outlook **42**

A. Appendix **43**
A.1. Clock Signal Forwarding 43
A.2. Simulation Results: Performance Measurement 44
A.3. Simulation Results: Reference Performance Measurement 45

Acronyms **46**

List of Figures **49**

List of Tables **50**

List of Listings **51**

Bibliography **52**

1. Introduction

1.1. Motivation

The entire world is controlled by computer chips. They are not always visible, but without them no modern car would drive today, no plane would take off, no washing machine would do laundry, and smartphones and tablets would not even exist. And all of this is only possible because the computer and chip industry has evolved rapidly over the past 60 years since Gordon E. Moore formulated his famous law and the first microprocessors emerged.

Moore's law, which the co-founder of Intel published back in 1965, held true until the late 2000s. It states that the packing density of the transistors on a microchip and thus the performance measured in million instructions per second (MIPS) doubles approximately every 18 months [1, p. 40]. This development meant that at some point entire computer systems with several processor cores, caches and other hardware components could be integrated on a single chip (die). These are referred to as Systems on Chips (SoCs). However, the predictions made by Moore's Law regarding the growth of transistor density on semiconductor chips were no longer valid as of around 2010. For this reason, and due to limitations in power density (end of Dinnard scaling) and multiprocessing (Amdahl's law), the doubling of processor performance has slowed down to approximately every 20 years [2, p. 5]. Nevertheless, billions are invested every year by the big technology companies in the research and development of new computer chips and technologies and therefore there is still rapid progress, always following the motto: smaller, faster, more efficient, cheaper. While in 2011 the Taiwanese chip manufacturer TSMC produced chips on the basis of a 28 nm process [3], the company will start producing the Apple M2 Pro chip in 3 nm technology in 2022 [4]. IBM has even announced the move to 2 nm technology in 2021 [5].

The question arises of how such a microchip, which integrates an entire computer system with one or more processor cores, is actually developed. And just as important, how the functionality and security of such an architecture can be verified.
An integral part of the answer to these questions is virtual prototyping. A virtual prototype is an executable software model of a hardware component, e.g., a central processing unit (CPU) or an interrupt controller. These virtual prototypes can be assembled into systems just like real hardware components. If such a system mirrors the full functionality of a real SoC to the outside world, it is referred to as a virtual platform (VP) [6]. VPs are usually implemented in parallel with the real hardware at an early stage of development and provide several advantages. On the one hand, it is not trivial to understand a highly

optimized system as a whole just from looking at it. Computer systems are very complex and it can therefore be difficult to identify and correct errors. Simulations with VPs are deterministic, can be started at any time without much effort, and most importantly, can be debugged more easily. On the other hand, implementing a virtual prototype is much cheaper than producing a prototype in real hardware at each stage of development. Another equally important application area of VPs and an active field of research is the safety and security verification of SoCs and microprocessors.
An example of non-commercial university research in this area is the work of the Chair of Electronic Design Automation (EDA) at the Technical University of Munich (TUM). There, the occurrence, the effect and the prevention of so-called soft errors are investigated. Soft errors are unpredictable, dynamic changes of memory cell states or states in sequential logic and can be also termed bit-flips. In contrast to hard errors, which are usually associated with a production defect or permanent damage to the hardware, soft errors can be caused, for example, by the random collision of radiation particles [2, p. 93], [7].
In order to investigate soft errors as described above, the Fault Injection Virtual Platform (FIVP) was developed at TUM-EDA, which is part of a so-called fault injection simulator. The FIVP models a generic, fully functional SoC with a CPU containing a single processor core. The special feature is that this processor core can be simulated at two different levels of detail. If one wants to examine the effects of a soft error at instruction or functional level, the core can be simulated by an instruction set simulator (ISS). An ISS takes the instruction to be executed on the simulated target instruction set architecture (ISA), e.g. RISC-V, and translates it just-in-time into an instruction that can be executed on the host ISA. The host is the computer on which the simulation runs. At TUM-EDA, usually the Extendable Translating Instruction Set Simulator (ETISS) is used for this purpose, which can be easily extended by plugins.
Although the simulation on instruction level is relatively efficient and fast, one often wants to examine errors more precisely and must be able to look into the processor core for this. The simulation level at which an ISS works obviously cannot provide this. Therefore, a model of a core at register transfer level (RTL) can also be inserted into the virtual prototype of the generic CPU. The RTL model describes the entire logic implemented in hardware in the core, so that the effects of soft errors can be examined much more precisely. Compared to instruction-level simulation, RTL simulation is more detailed, but for this reason also much slower. This can make a significant difference in simulation time when executing several million instructions that a test program may have. Furthermore, with fault injection, the entire execution up to the injection point of the soft error is of no interest. If the error has no effect or is masked, the execution after the error is also uninteresting. For this reason, it would make sense to simulate at instruction level from the beginning, then switch to RTL shortly before the error is injected, and finally switch back to instruction level if the error has no further effects.

The objective of this work is to develop such a runtime switch behavior for the described multi-level fault injection simulator. The initial focus will be on the "switch down" from the instruction-based simulation to the RTL simulation. In this context, the fault injection

will be disregarded at the beginning, since it has not yet taken place at the time of the "switch down". Instead, an algorithm is to be found and implemented that can perform the switch to RTL on any instruction by jump-starting an instance of the CPU's virtual prototype with the RTL core, initializing the core, mapping the states from the ISS to the RTL core, and then letting it run. Thereby, the switch behavior should be implemented as general as possible, so that it can be applied to different cores or even ISAs. In addition, it must be taken into account that possibly states of the RTL model, which the ISS does not model, must be calculated and initialized separately.
In order to be able to verify the functionality of the entire switch behavior, a tandem simulation is to be developed first, in which a reference RTL core is simulated in parallel with the ISS. Subsequently, the switch behavior shall be implemented. Afterwards, fault injection can be performed and an algorithm can be found to decide when a fault has been masked or is no longer relevant to the following program flow. However, this and the following switch back would exceed the scope of this work. Nevertheless, if the injected error has been masked and the RTL core still behaves correctly according to the ISA, the switch back should be the more trivial of the two switches, since the ISS does the same as the RTL simulation on just a more abstracted level.

In the following, considerations and existing implementations by other researchers are discussed and compared with the intention of this work. The next chapter will then describe the background information needed to understand the present work. VPs, ETISS, the RISC-V ISA, and the FIVP will be discussed in more detail. Afterwards, the runtime switch behavior is worked out conceptually in Chapter 3, before Chapter 4 deals with the implementation. This is done in form of a CPU multiplexer component. Finally, the performance increase with respect to the simulation time is evaluated experimentally and the resulting potentials are discussed.

1.2. Related Work

The runtime switch behavior for multi-level simulators has already been studied and described by a few institutions. For instance, Xing et al. [8] propose a "tandem simulation (or RTL co-simulation with ISA simulator)" and mention a scenario where the RTL simulation is jump-started. This is realized with the help of a "cold start" map, which is part of a so-called refinement map. In response to a personal inquiry, Xing stated that this refinement map was implemented manually and not optimized. According to him, this is sufficient for their application scenario, since their problem satement is to verify that the RTL model sufficiently refines their ISS. Accordingly, no switching back is mentioned or envisioned in their scenario. However, the refinement map developed by Xing based on the work of Huang et al. [9] is a good reference for the present work. Huang et al. [9] likewise describe a verification scenario in a multi-level tandem simulation, but without mentioning a switch behavior. Nevertheless, they propose to use shared memory in the simulation, just as the simulation in this work will do.

In 2015, Aarno and Engblom [10] already proposed a multi-level switch behavior for a VP, which they call gear-shifting. It can switch from an abstracted "Simics model to a detailed model" only in certain checkpoints. The "detailed model" is not described in detail, but is likely to correspond to an RTL model. They also describe for the first time a switch back to instruction level, but remark that switching back is not always trivial either, since in modern processors with multi-stage pipelines it is not always clear which instruction is being executed at the moment. As a consequence, they suggest that instead of switching back, it would be more efficient to stop the detailed simulation and reset the higher-level simulation to the point in time before the "switch down". Then, this simulation runs fast up to the point to which the detailed simulation had already come, and continues from there. The author of the present work has several comments on this approach. First, the switch behavior developed in the present work should allow switching simulation levels on every instruction, not just at specific checkpoints. Second, the author suggests jumping back and continuing execution from the instruction that was last committed. This should work at least for in-order-execution processors. Third, resetting the simulation is not straightforward if both simulations share memory and a VP with peripheral components. Aarno and Engblom do not describe how they handle the problem of peripheral timings and events that occur during the period that is reset. These would also have to be reset or repeated too, which the author of this work considers to be difficult. Furthermore, in the fault injection application scenario of this work the detailed model is manipulated and if this has consequences for architectural states, it should also be visible in the more abstracted simulation after the switch back. Accordingly, the idea of Aarno and Engblom, who describe a switch back for the first time, is not practical for this work.

A dynamic switch behavior for use in a fault injection scenario, where both a "switch down" and a "switch up" are possible, was proposed by Mueller-Gritschneder et al. [11]. In 2018, they presented the multi-level extension of the instruction set simulator ETISS, which they call ETISS-ML. This "achieves close-to-RTL-accurate fault injection simulation

results with close-to-ISS simulation performance" [11]. The simulation flow proposed by Mueller-Gritschneder et al. is shown in Figure 1.1. It illustrates the "switch down" from the ISS level to RTL, the application of fault injection, and the switch back to the ISS. The present work is based on this approach and the described simulation flow. Independently of ETISS-ML, a runtime switch behavior for the FIVP is to be developed, which allows to switch arbitrarily between instruction-level and RTL simulation.

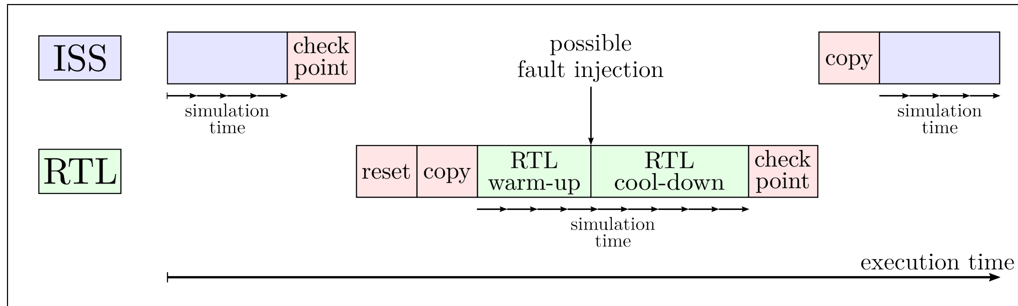


Figure 1.1.: Multi-level simulation flow suggested by Mueller-Gritschneder [11]

A problem of the ETISS-ML approach is that a pipeline for fault injection and a reference pipeline are simulated together in one core. For the switch operations and fault injection, an elaborate and precise definition is necessary of which states should be mapped to which other states and which micro-architectural signals and states of the two pipelines must be compared. This is due to the fact that by duplicating the pipeline a fine-grained reference model is used. In the context of this work, an approach is to be developed in which two cores (one for fault injection and one as a reference) are run side by side and the comparison can be made either at micro-architectural level or at architectural level. This could be referred to as a more coarse-grained reference model, which does not require such complex and architecture-specific definitions and comparisons. The difference between ETISS-ML and the principle to be developed in this work is illustrated in Figure 1.2. Furthermore, RISC-V will be used as the primary simulation target ISA instead of OpenRISC, although the algorithm itself is universally applicable to other ISAs.

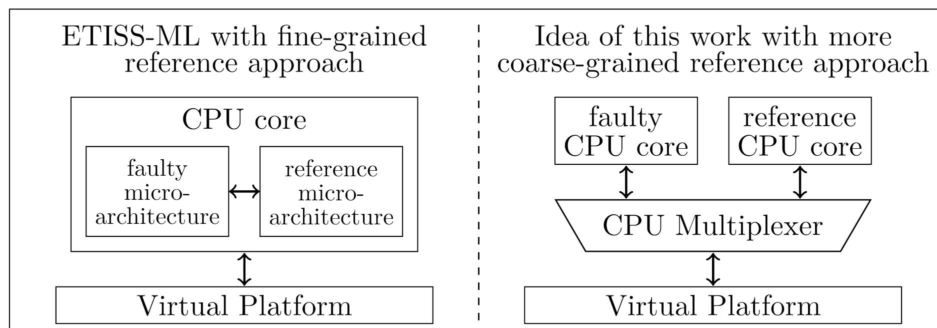


Figure 1.2.: Difference between ETISS-ML and the idea of this work

2. Background Information

2.1. Virtual Prototyping

A virtual prototype is an executable software model of a hardware component that can be run on a host computer and mirrors the functionality of the hardware component [12, p. 17]. Different virtual prototypes can interact with each other and be assembled into a larger system. If this virtual system consisting of several virtual components represents a real computer system or respectively a system on chip (SoC), it is called a virtual platform (VP). An example of such a VP is the FIVP discussed in Section 2.7.

When implementing a virtual prototype a certain level of abstraction must be defined. For the functionality of a simulated component and the entire system, it is not necessary to model the hardware down to the smallest detail. For example, cycle-based state machines, which are implemented in real hardware with registers and circuit lines but are not externally visible, can be abstracted in the model by simple variables. By contrast, registers that are also externally visible in the real hardware and for example represent interfaces must also exist in the virtual prototype and have the correct value at every point in time.

Traditionally, it was the case in large technology companies that the hardware was developed first and then a hardware prototype was available shortly before the start of production. At that point, the software developers could start implementing the firmware and drivers and test them with the prototype. Often, the availability of a first hardware prototype was only shortly before the product release date, so that the software development and especially the tests on the hardware had to take place under time pressure.
Virtual prototyping fundamentally changed this process. Nowadays, the implementation of virtual prototypes and platforms already starts alongside the hardware development. Due to the abstraction explained in the previous paragraph, these require a much shorter development time than the real hardware. De Schutter mentions a time difference of 9 to 12 months [12, p. 21]. This enables software developers to start developing, debugging and validating system software much earlier, since they can do the latter on the virtual prototypes instead of the real hardware prototypes. In addition to the time availability described above, the use of virtual prototypes for debugging and testing during the development of firmware or drivers for new hardware has further advantages:

- **Visibility:** Virtual prototyping allows to look anytime into the hardware component with any degree of precision. This means that during simulation it is possible to

look into every single hardware block and to check the status of every register and signal, as long as it is not abstracted by the simulation. This is not possible with real hardware because a chip produced at nanometer scale can not at all be opened to measure signals or register values. Furthermore, the visibility allows parallel debugging. Since the simulation can be interrupted at any time and it is always clear at which point of the execution the system software and the hardware simulation are, one can debug both in parallel. This means, it can be seen what the system software, e.g., firmware, would like to do and what the hardware does in reality. This way errors can be found much faster than by debugging with real hardware.

- **Control:** The software developer has full control over a system of virtual prototypes. The simulation can be paused at any time, then memory can be examined or manipulated before the simulation can be resumed without trouble. This is also not possible with real hardware.
- **Determinism:** Virtual prototypes are deterministic. Determinism refers to the property that a program or an algorithm always behaves the same given the same input, i.e., always produces the same output. For a virtual prototype this means that it always behaves exactly the same with the same prerequisites such as the same instructions or the same memory configuration. This is particularly useful for testing, since errors can be reproduced as often as desired. In test environments for real hardware, this is not the case, since there can be many external influences, such as radiation particles that interfere with the electronics.
- **Portability:** Assuming that the team developing the hardware, the production facility for the hardware prototype, the team responsible for testing and the firmware developers are located in completely different places in the world. In this scenario, hardware components very often have to be shipped around the world. Virtual prototypes can be easily shared over the internet, which makes the whole process and deployment much easier.
- **Fault Injection:** The controllability and visibility properties described above allow not only testing the correct case but also active fault injection. The injection itself and the evaluation of the effects are significantly more difficult in real hardware. An example of using a VP for fault injection is the Fault Injection Virtual Platform (FIVP) described in Section 2.7.

In summary, it can be said that virtual prototyping is used in many industrial companies engaged in hardware and software development. They use virtual prototypes to develop and debug software in parallel with hardware development and to validate the behavior of the overall system. This shortens the development time, improves the quality of the product and thus maximizes profit [12].

2.2. SystemC

The de facto standard for the implementation of virtual prototypes and thus VPs is SystemC TLM-2.0. It is defined in the IEEE standard (IEEE Std. 1666-2011) [13] and can be used to model the behavior of hardware systems and components at a certain level of abstraction. Therefore, it is ideal for implementing virtual prototypes.
TLM stands for transaction-level modeling and is as an extension of SystemC also defined in the SystemC standard. It provides standardized interfaces to simplify the modeling of data transactions between virtual hardware blocks. The next subsection describes the simulation kernel and the core language concepts of SystemC, before TLM-2.0 is discussed in Section 2.2.2.

2.2.1. SystemC Simulation Kernel and Core Language Concepts

A SystemC application is an event-based, discrete real-time simulation. Therefore, the heart of the SystemC class library is a private simulation kernel that includes a dedicated scheduler. This can execute processes within the application. The execution of a SystemC application is now split by the kernel into two phases, the elaboration phase and the simulation phase. Both are shown in Figure 2.1 and briefly explained below.

The first phase is the elaboration phase. In this phase the kernel instantiates all SystemC processes and builds up a module hierarchy by instantiating and connecting all modules. Modules can be instantiated in SystemC by calling the constructor `sc_module()`. They can be built hierarchically from other modules and contain ports which are connections to the outside. These can be thought of as pins of real electronic components. There are three classes derived from the `sc_port` class, namely `sc_in`, `sc_out` and `sc_inout`. The latter can be used as both input and output, the others only as input or output. Modules are connected via the described inputs and outputs by using so-called primitive channels. The most commonly used channels are the `sc_signals` derived from the `sc_prim_channel` class. They forward values like wires in real circuits. Besides `sc_ports` there are also `sc_exports`, which are made to connect modules not to the outside but to parent modules. It is important to note that elements of the type `sc_object` may only be instantiated during the elaboration phase. This is the case because SystemC does not support dynamic creation or modification of objects during simulation time. All elements of the following classes or classes derived from them are `sc_objects`:

- `sc_module`
- `sc_port`
- `sc_export`
- `sc_channel`

At the end of the elaboration phase the time resolution must be defined. It is defined globally and can also not be modified at simulation time.

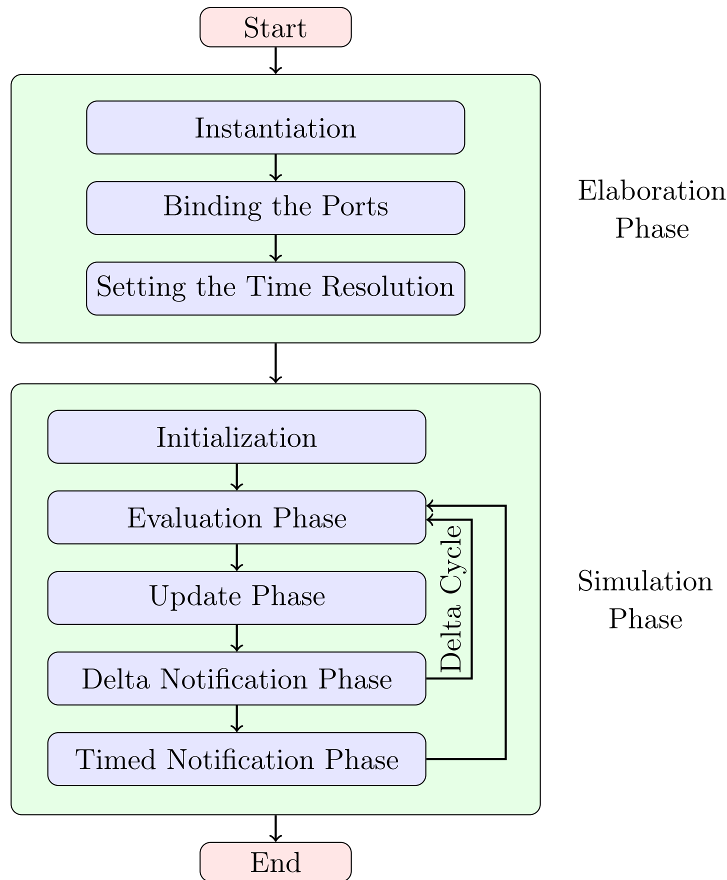


Figure 2.1.: Phases of a SystemC simulation

After the elaboration phase, the simulation phase is started. Figure 2.1 shows the five steps executed in the simulation phase. The first one is the initialization of the simulation. Thereby, the update phase, which is described below, is performed once. Then, all instantiated SystemC processes are added to the pool of runnable processes. Afterwards, the delta notification phase is run once. After the initialization phase, the phases are performed as shown in Figure 2.1. First, an evaluation phase is performed in which the current values of all signals, ports, etc. are calculated. These are then updated in the update phase. Since SystemC is event-based and sensitivity dependencies can be defined, it can be the case that new events are triggered due to the previously executed updates or that signals depending on previously updated signals have to be updated themselves. For this reason, a delta cycle is executed next. The most important thing about delta cycles is that the simulated time does not advance. As the loop in Figure 2.1 shows, the simulation kernel then returns to the evaluation phase. It checks whether dependent values have changed due to the mentioned events or sensitivities. Sensitivities will be explained when `sc_threads` and `sc_methods` are introduced. The values are updated in the following update phase if necessary. This procedure is done until no values change in the current time step due to any dependencies. Then a real time step is made, i.e., the simulation

time is increased to the point when the next event from the event queue must be handled. The whole simulation runs until the simulation time has expired or the simulation is manually stopped by the user by calling `sc_stop()`. [13, p. 12ff]

One of the most important features of SystemC is that parallelism can be simulated. Therefore, it is important to know that there are processes in SystemC. A process is an execution line of instructions and several processes can run concurrently. At least this concurrency is simulated by executing all processes in a delta cycle one after the other. However, in SystemC there are three kinds of processes: `sc_thread`, `sc_cthread` and `sc_method`. An `sc_thread` is executed exactly once and then terminates. Therefore, endless loops are often used in threads. `Sc_cthreads` are clocked threads, i.e., there is a clock specified that triggers it. Finally, `sc_methods` are processes that are not executed only once and then terminate, but are executed each time the event to which they are sensitive is triggered.
Processes can be triggered, e.g., by `sc_events`, clock events (`sc_cthreads`), or other events if a sensitivity list is defined for them in the constructor of the module. In the constructor `sc_threads` and `sc_methods` can be defined as sensitive to any events, e.g., a function call or a rising edge of a port value. This means that every time this event occurs the simulation kernel calls the thread or method that is sensitive to this event. Sensitivities can be defined after the definition of the threads or methods with the sensitivity operator: `sensitive << event;`.
Moreover, `sc_threads` can be paused by calling `wait()`. Then, they are continued when an event occurs they are statically sensitive to. If `wait()` gets the parameter `SC_ZERO_TIME`, the process is continued in the next delta cycle. Alternatively, it can be waited for a specific event. This also allows process synchronization. `Sc_methods`, to the contrary, cannot contain `waits` by definition.

2.2.2. Transaction Level Modeling (TLM-2.0)

As introduced in the previous section, transaction-level modeling (TLM) is an extension of SystemC and also defined in the SystemC standard [13, p. 413ff]. TLM components are assembled from SystemC elements and standardize and simplify the simulation of data transfer operations between TLM components through so-called sockets. The latest version is TLM-2.0.

Before the main elements of TLM-2.0 can be discussed, the two TLM coding styles must be introduced:

- **Loosely-timed:** The loosely-timed coding style uses blocking transport functions to model data transfers. Here, exactly two timing points of the simulation time are used. The first timing point is the start of the transport request, i.e., the call of the function `b_transport()`. The second point, to which both the sender and the receiver must adhere, is the beginning of the response. These two points can be in the same simulation time step or in different and are sufficient to guarantee

a reliable, more or less time-accurate communication without the need for explicit synchronization at each transmission. Therefore, the loosely-timed coding style is the optimal choice for the implementation of virtual prototypes and VPs [13, p. 417]. It allows the modeling of bus systems, timers and interrupts and is accurate enough run a firmware on the VP or even boot an operating system (OS) on it. At the same time, the loosely-timed coding style requires only a minimal number of synchronization points, so that the performance of the simulation is not noticeably harmed by the TLM communication. In addition, this coding style supports temporal decoupling. This means that processes may run up to a certain point in time - e.g., a global quantum or a waiting point for an event - ahead of the simulation time. This again can increase the performance.

- **Approximately-timed:** The second coding style is the approximately-timed coding style. It is more accurate in time than the loosely-timed style but requires four timing points. These are the beginning and the end of the transport request and the beginning and the end of the response. Due to the demand for a higher time accuracy, no temporal decoupling is possible here. This coding style is suitable for architecture exploration and performance analysis.

As described above, the loosely-time coding style is the optimal style for modeling virtual prototypes and VPs. For this reason, the use of this coding style is always implicitly assumed in the following.

In general, TLM-2.0 consists of a set of core interfaces, the global quantum, initiator and target sockets, the generic payload and base protocol, and some utilities. Each of these five main components is briefly explained below.

- **Core interfaces:** TLM-2.0 provides four different transport interfaces: the blocking and the non-blocking transport interface, the direct memory interface, and the debug transport interface. As indicated above, with the loosely-timed coding style the blocking transport interface is primarily used. It provides the pure virtual function `b_transport()`, which performs the transaction between two modules. The function requires two arguments. The first one is a non-constant reference to a TLM generic payload, which will be introduced below. The second argument is a timing value indicating the start and end of the transaction.
  The non-blocking transport is intended to support the approximately-timed coding style and is therefore not explained further here. The direct memory interface (DMI) is a specialized interface that allows direct access to memory of the target. By bypassing the normal path through the individual connection components, DMI can accelerate regular memory transactions in loosely-timed simulations. The debug interface allows transactions to be performed without delay by, so to speak, bypassing the simulation.

- **Global quantum:** The global quantum is an `sc_time`, which is defined globally for all transactions. The advantage of this is that all transactions use the same time

base and that, e.g., in the case of temporal decoupling a fixed limit is defined up to which a process may run ahead of the simulation.

- **Initiator and target sockets:** Sockets are a central element in TLM. Via these sockets modules can exchange data because they contain `sc_ports` of the type of the interfaces defined above. There are `tlm_initiator_sockets` and `tlm_target_sockets` which can be bound together. An initiator socket initiates a data transaction that the target socket receives and then calls a previously registered callback function. The initiator socket then receives the result of this on the backward path. Figure 2.2 illustrates how `tlm_initiator_sockets` and `tlm_target_sockets` are connected using the interfaces described above. This connection can be used to send data packets which have already been introduced above as `tlm_generic_payload`.

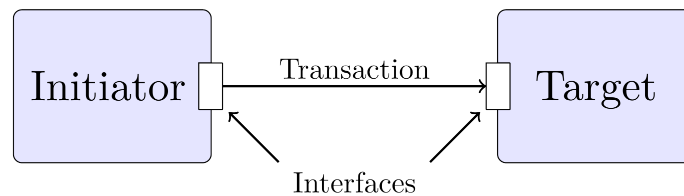


Figure 2.2.: Transaction between a TLM initiator socket and a target socket

- **Generic payload and base protocol:** A generic payload (GP) is an object that typically contains attributes such as a `tlm_command`, an address, data, and a response status. Usually, only a reference to a GP object is passed from the initiator to the target. This contains a command, e.g., the `tlm_command` read, and a memory address. The data from the specified address is then read by the target and written into the designated field of the GP before the response status reports back that the transaction has been completed. The GP is closely linked to the TLM base protocol, in which rules for communication are defined. In addition to the communication rules, the protocol also defines the behavior of `tlm_sockets`, the described transport interfaces, and the transmission phases request and response.

- **TLM utilities:** To the utilities namespace belong a set of classes that contribute to convenience for the programmer and consistency of coding style. For example, they provide the convenience sockets like the `simple_initiator_socket` and the `simple_target_socket`. These are derived from the original sockets and simplify, among other things, the registration of callback functions.

2.3. SystemVerilog

SystemVerilog is a unified hardware design, specification, and verification language defined in the IEEE standard 1800-2017 [14]. It is an extension of the hardware description language (HDL) Verilog and is designed to describe hardware at register transfer level (RTL). It also allows convenient testbench verification among other things through provided application programming interfaces (APIs) and direct programming interfaces (DPIs).

The most important difference between an HDL and a programming language the reader should know is that an HDL describes the structure and behavior of a hardware logic as a whole, while a programming language is used to program a sequence of instructions a central processing unit (CPU) can execute in the given order.

In order to understand the present work it is not necessary to know the structure and the syntax of SystemVerilog. Therefore, a comprehensive description of SystemVerilog is omitted here. The reader should note that the core model at RTL used as a target in this work is the RI5CY core, which is described in Section 2.6. It is implemented in SystemVerilog and is converted to executable C code using the Verilator tool described in the next section.

The only change made to SystemVerilog code in the context of this work is the implementation of DPIs, which have already been introduced above. Therefore they will be briefly described here. A DPI is an interface between SystemVerilog and another programming language, for example C. This means that functions implemented and exported in SystemVerilog can be called in C code. Conversely, functions implemented in C can be imported and called in SystemVerilog code. This feature is used in the present work to read or manipulate states and variables resembling sequential (flip-flops) or combinational (nets, signals) logic of the RI5CY core model. Such a state can be, e.g., the instruction pointer (IP) in the decode stage of the processor or the value of a general-purpose register (GPR).

2.4. Verilator

Verlilator is an open-source tool that converts hardware designs defined in the HDLs Verilog or SystemVerilog into C++ or SystemC models. Since code is translated from one language to another, Verilator must be called a compiler. The verilated RTL (VRTL) models can then be compiled into an executable binary using a C++ compiler [15]. The described compilation flow is visualized in Figure 2.3.

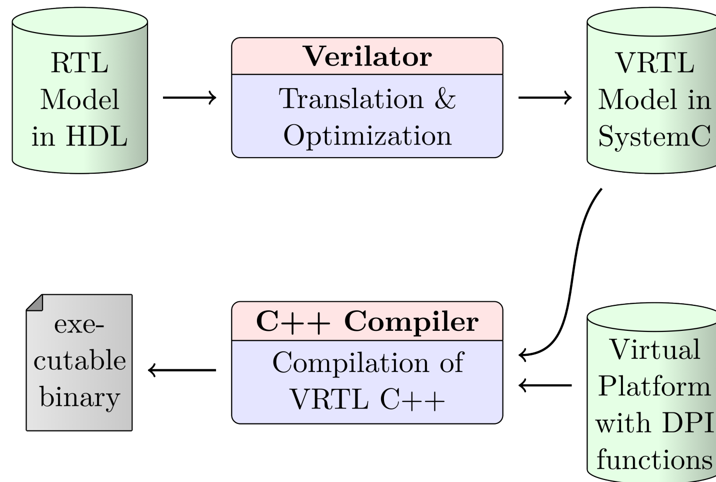


Figure 2.3.: Verilator flow

Veripool, the developing organization behind Verilator, states that a compiled Verilog model executes even in a single thread up to 10 times faster than standalone SystemC [16]. This is achieved mainly through optimizations during Verilator's synthesis to C++, which for example include the flattening of module hierarchies and the abstraction of data types.

VRTL models can interact with other SystemC components in two ways. Firstly, the VRTL model has SystemC ports as inputs and outputs to which SystemC signals can be bound. Secondly, Verilator also supports the SystemVerilog DPI import and export instructions. This means that SystemVerilog functions can be called directly from C++ code and vice versa.

2.5. Extendable Translating Instruction Set Simulator (ETISS)

The Extendable Translating Instruction Set Simulator (ETISS) is a C++ instruction set simulator (ISS) developed by the Chair of Electronic Design Automation (EDA) at the Technical University of Munich (TUM). It simulates instruction execution by translating the binary code for a given target instruction set architecture (ISA) into C code, which is then assembled into blocks. Thereafter, each block is compiled into binary code for the host computer's ISA and executed. A strength of ETISS is, as the name already says, that it can be extended very easily by so-called plugins. ETISS supports several target ISAs. However, in the context of this work only the RISC-V architecture is being used.

2.5.1. Architecture of ETISS

ETISS consists of two main components, as shown in Figure 2.4. One is the initializer and the other is the CPU core. The Initializer sets up ETISS and ensures that all components necessary for the CPU core are available and usable. This includes a just-in-time compiler (JIT), the definition of the target ISA in the `CPUArch` object and any number of ETISS plugins. The definition of the target ISA is necessary so that the binary code can be translated into host-compilable C code by the translation plugin. The JIT is the component which subsequently compiles the blocks of C code for execution on the host machine. ETISS supports standard C compilers, e.g., GCC, TCC or LLVM compilers. The plugins are explained in the next subsection and some examples are provided.

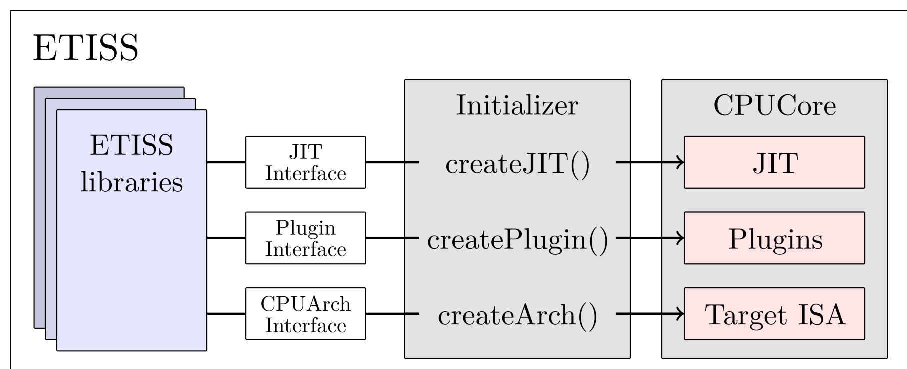


Figure 2.4.: Etiss architecture [17]

When the simulation is started, all plugins are initialized and an endless execution loop is entered. This loop is shown in Figure 2.5 and is entered through the green statement block. Unless an exception occurs, the translation plugin is directly tasked with delivering the next block of translated C instructions. To do this, it gets the current IP, then reads the next binary instructions without delay via the debug interface and translates them into a block of C instructions. Next, this block is executed, as shown by the bottom blue statement block. Afterwards, it is checked if an exception has occurred. With such an exception the simulation can be terminated, for example. If the exception handling was

successful, possibly other plugins are executed and the loop is continued from the green statement block.

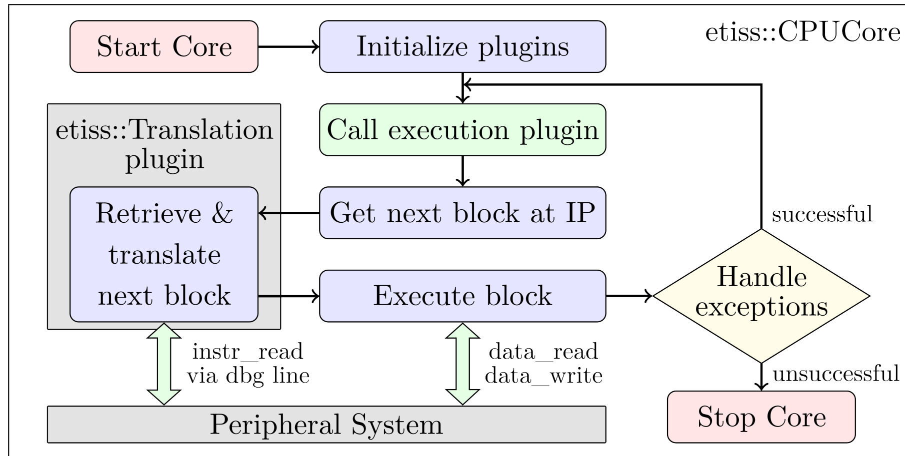


Figure 2.5.: Etiss core execution loop [18]

2.5.2. Plugins for ETISS

Plugins are a simple and comfortable way to add functionalities to ETISS. They are included, registered and initialized, and then called once per cycle in the execution loop of the ETISS CPU core. As already mentioned above the translation of the binary code into C code is done by a plugin. Likewise, the JIT that compiles the code into host-executable instructions is included as a plugin. In the following other application areas of plugins are listed:

- interrupt listening and handling
- reset and termination of the CPU
- logging
- connection to a debugging server, e.g., a GDB server
- timer implementation

2.5.3. Virtual Core Structure in ETISS

The ETISS CPU core contains a simple structure that is a virtual representation of the core, a so-called VirtualStruct. This is of course highly ISA dependent and must therefore be defined in the included architecture implementation. In the VirtualStruct, fields can be defined which are addressed by their name and are connected with the real register values in ETISS. This could be, e.g., the IP, a GPR or a CSR register. Via the VirtualStruct the values in ETISS can be read and written at any time.

2.6. RI5CY RTL Core

2.6.1. RISC-V Instruction Set Architecture Overview

RISC-V is a free and open reduced instruction set computer (RISC) ISA developed by the University of California, Berkeley. It is the fifth generation of RISC architectures and a simple and widely used load-store ISA. As of 2022, it is used in academia and industry[1], and Hennessy and Patterson even use the integer core ISA of RISC-V as an example architecture in their book [2]. In the following the main characteristics of the RISC-V ISA are listed.

- **Variants:** RISC-V provides a 32-bit, a 64-bit, and a 128-bit instruction set as well as a variety of extensions for features like floating point arithmetic.
- **Registers:** RISC-V has 32 general-purpose registers (GPRs) and 32 floating-point registers (FPRs). In addition, it has multiple control and status registers (CSRs).
- **Memory access:** RISC-V can access memory only with load or store instructions.
- **Addressing Modes:** RISC-V offers three addressing modes, which are Register, Immediate (for constants), and Displacement. The latter two have 12-bit fields to define the immediate and the displacement respectively.
- **Operands:** Like most ISAs, RISC-V supports operands of 8-bit, 16-bit, 32-bit, 64-bit, 32-bit floating-point (single precision), and 64-bit floating-point (double precision).
- **Operations:** RISC-V supports a list of simple operations of the classes: loads and stores, ALU operations, branches and jumps, and floating-point operations. Hennessy and Patterson summarize the set of operations at [2, p. 15f].
- **Encoding:** If the compressed instruction set extension is not used, all RISC-V instructions are 32-bit, which simplifies instruction decoding.
- **Control flow instructions:** RISC-V provides two jump instructions (jump and link and jump and link register) and a variety of branch instructions. All branches are conditional.

All properties can be incorporated into the design of a RISC-V processor that is capable of executing RISC-V instructions. For efficiency reasons, this can be done in the form of a pipeline in which an instruction passes through multiple stages. There are many concrete implementations of the RISC-V ISA in real hardware. One of them is the RI5CY implementation, which is used in the present work and described in the following section.

[1] More than 60 companies have joined the RISC-V foundation, including AMD, Google, HP Enterprise, IBM, Microsoft, Nvidia, Qualcomm, Samsung, and Western Digital [2, p. 12]

2.6.2. RI5CY Implementation

RI5CY is an open-source implementation of a 4-stage in-order 32-bit RISC-V processor core which was developed by the ETH Zurich and the University of Bologna in context of the Parallel Ultra Low Power (PULP) project. In the meantime, the RI5CY core has been licensed by the OpenHardwareGroup under the name CV32E40P. For the sake of comprehensibility, this work will continue to refer to it as the RI5CY core. It supports the RV32I Base Integer Instruction Set of RISC-V and several extensions like the RV32C Standard Extension for Compressed Instructions, the RV32M Integer Multiplication and Division Instruction Set Extension, and some PULP specific extensions which are not discussed in this work [19]. Figure 2.6 shows the block diagram of the RI5CY processor.

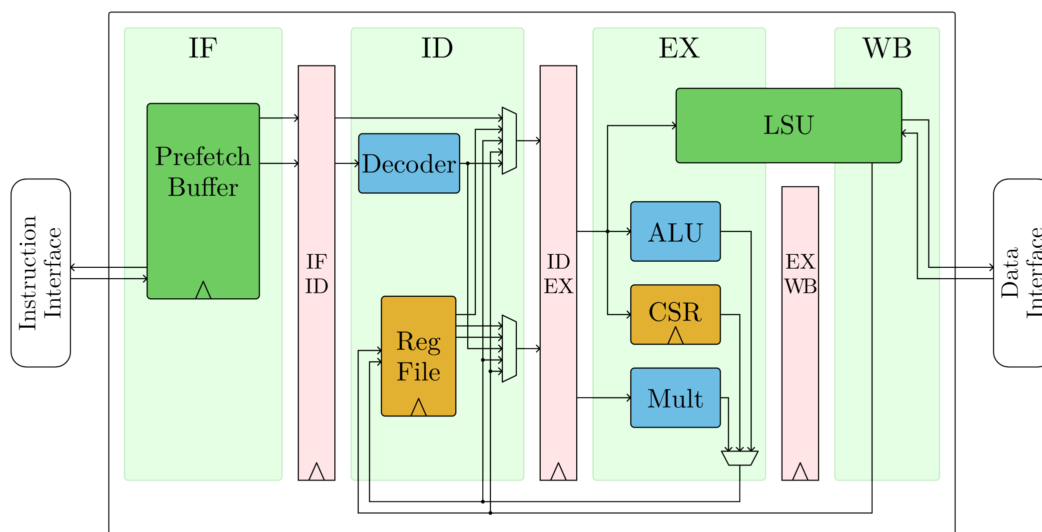


Figure 2.6.: Block diagram of the RI5CY processor [19]

The four pipeline stages of the RI5CY processor and their tasks are listed below.

IF The *instruction fetch* stage contains a prefetch buffer which fetches instructions from the instruction memory or cache. It is a 128-bit cache line buffer which means that it can store four words. The fetching starts at address 0x80.

ID In the *instruction decode* stage the instructions are decoded in data and control signals. This stage also contains the register file explained below and in Table 2.1.

EX In the *execution* stage the instructions are actually executed in the respective execution unit, e.g., the arithmetic logic unit (ALU) or the optional floating-point unit (FPU). In addition, the CSRs are written in this stage.

WB The *writeback* stage contains the load-store unit (LSU), which manages the data memory interaction. In addition, values are written back to the registers in the ID stage or forwarded to the EX stage.

All pipeline stages of the RI5CY core are independent of the previous stage. The independence means that each stage can finish execution regardless of whether the previous stage is stalled or not. If a stage and the following stage are in the ready state, the instruction is moved to the next stage on the next clock edge. [19]

The RI5CY core can be configured to contain either a flip-flop based or a latch-based register file in the instruction decode stage. In the context of this work a flip-flop based register file is used. Also, the FPU is not used, so that no FPRs but only GPRs are needed. This reduces the memory size of the register file. The resulting register file with the 32 GPRs is shown in Table 2.1. Furthermore, it should be noted that the RI5CY core is used exclusively in machine privilege mode in this work.

Register	Name	Use
x0	zero	The constant value 0
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function arguments/return values
x12-x17	a2-a7	Function arguments
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporaries

Table 2.1.: RISC-V integer register file with 32 GPRs [20]

2.7. Fault Injection Virtual Platform (FIVP)

The FIVP is a VP developed at TUM-EDA, where it is used to study the effects of soft errors injected into processor cores. It models a fully functional SoC, which is however kept as minimalistic as possible, since it is used exclusively for the described fault injection of studied CPUs. Figure 2.7 shows the various components and the wiring of the FIVP. It has one system bus to which a CPU is connected via two TLM sockets. The instruction line of the CPU is connected to one of the sockets and the data line is connected to the other. Besides the CPU there are a few peripheral components that are also connected to the system bus. The most important component is the memory (MEM) block, into which the program to be executed is loaded as an ELF file at the start of the simulation. There is also an external timer block, a universal asynchronous receiver-transmitter (UART) controller, a core-local interrupt controller (CLINT) and a platform level interrupt controller (PLIC). The CLINT is responsible for handling software interrupts or internal timer interrupts,

while other peripherals of the VP that can trigger external interrupts are connected to the PLIC.

The unique feature of the FIVP is that different cores can be inserted into the CPU. Either the CPU can be a SystemC wrapper for an ETISS core (see Figure 2.5). Since ETISS is an ISS, the simulation is in this case done at the instruction level. Alternatively, a VRTL model of a RISC-V core can be inserted into the CPU. Then, the simulation is done at RTL. As of 2022, FIVP supports the CVA6 (Ariane), CV32E40P (RI5CY), and CV32E40S (secure derivative of RI5CY) cores. In the context of this work, only the RI5CY core is used, which was introduced in Section 2.6. The output signals of the VRTL core are translated into TLM transactions in an instruction bridge and a data bridge, so that they can be sent to the system bus.

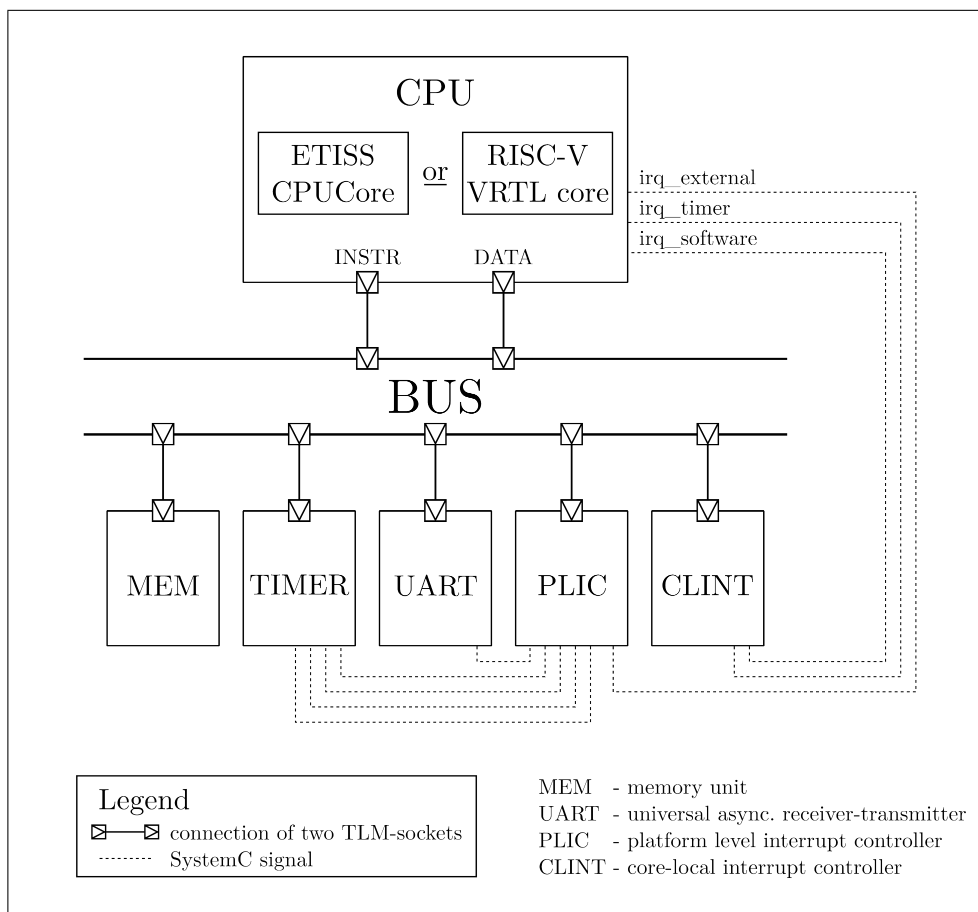


Figure 2.7.: Block diagram of the FIVP

3. Conceptional Development of a Runtime Switch Behavior for Virtual Platforms

This chapter defines what a runtime switch behavior for virtual platforms (VPs) is and discusses alternatives for implementing it. Based on the concepts developed in this chapter, the concrete implementation of a switch behavior that can be used for example in the previously presented Fault Injection Virtual Platform (FIVP) is described in the following Chapter 4.

3.1. Definition of a Runtime Switch Behavior

In the context of this work, the term runtime switch behavior describes the ability to switch during the simulation runtime of a VP between different simulation levels, i.e., levels of simulation detail. More precisely, this refers to the simulation level of the CPU core. As introduced in Section 2.7, in the FIVP there is the possibility to use either a CPU wrapping an instruction set simulator (ISS) like ETISS or a CPU with a verilated RTL (VRTL) core model. Table 3.1 lists the properties and advantages of both simulation levels.

CPU wrapping an ISS	**CPU with a VRTL core**
• simulation on instruction level • behavior of the CPU imitated as seen from the outside • abstraction: architectural states (like GPRs and CSRs) are available but no details inside the core are simulated • high simulation speed (in MIPS)	• simulation at register transfer level (RTL) • hardware inside the core simulated • detailed simulation/investigation possible • useful for micro-architectural fault injection • low simulation speed (in KIPS)

Table 3.1.: Properties and advantages of different simulation levels

Usually the user has to decide before simulation time on which level the simulation should run. Either, the more performant simulation at ISS level can be chosen, which

only represents the architectural behavior of the CPU. Alternatively, a more detailed simulation at RTL can be run, which is slower. For the purpose of fault injection, the latter is normally chosen, since faults are to be injected and investigated at the micro-architectural level as well [21]. However, often only short sections of the simulation are interesting for research. In the fault injection scenario, everything up to the point of the injection is uninteresting and then follow a few thousand cycles which are to be investigated. Subsequently, everything is less relevant again because the fault has either had catastrophic effects, the CPU is still behaving like a CPU, or the fault has been completely obscured. Nevertheless, in the existing approach, all passages have to be simulated at the same simulation level and thus speed.

A runtime switch behavior can significantly shorten the simulation time by combining both simulation levels and their strengths. The simulation can run at ISS level from the simulation start until the point of interest, e.g., fault injection. Then, a switch to the core at RTL can be performed. Next, the investigations are done at RTL and maybe it is possible later to switch back to the fast ISS level. This procedure was already shown in Section 1.2 when the work of Mueller-Gritschneder [11] was discussed and is shown again in an abstracted form in Figure 3.1.

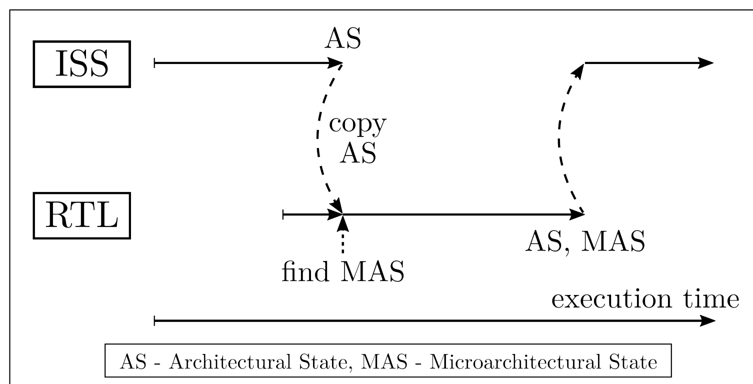


Figure 3.1.: General idea of a runtime switch behavior

As indicated in the figure, the switch from ISS level to RTL is not trivial. On the one hand, the ISS' architectural states (AS), such as register values, must be mapped to the states of the RTL. On the other hand, the RTL core must be started, it must reach a valid state and, if necessary, values must be found for states that are not present in the ISS. These are described as micro-architectural states (MAS) in Figure 3.1.

Figure 3.2 shows how the runtime switch behavior could be used in the context of fault injection. First, the simulation is started again at ISS level prior to switching to the RTL. In this illustration, the RTL core is run for a few cycles after the reset and the mapping of the states to make sure that it is in a valid state. This is called the warm-up phase. Next, a clone of the RTL core is created, into which a soft error is injected. This way there is a faulty core and a non-faulty reference core that can be compared to study the effects

of the injected error. After the fault injection investigation is finished, it can either be switched back to the ISS level or the simulation can be finished at RTL. Figure 3.2 shows the scenario with the switch back.

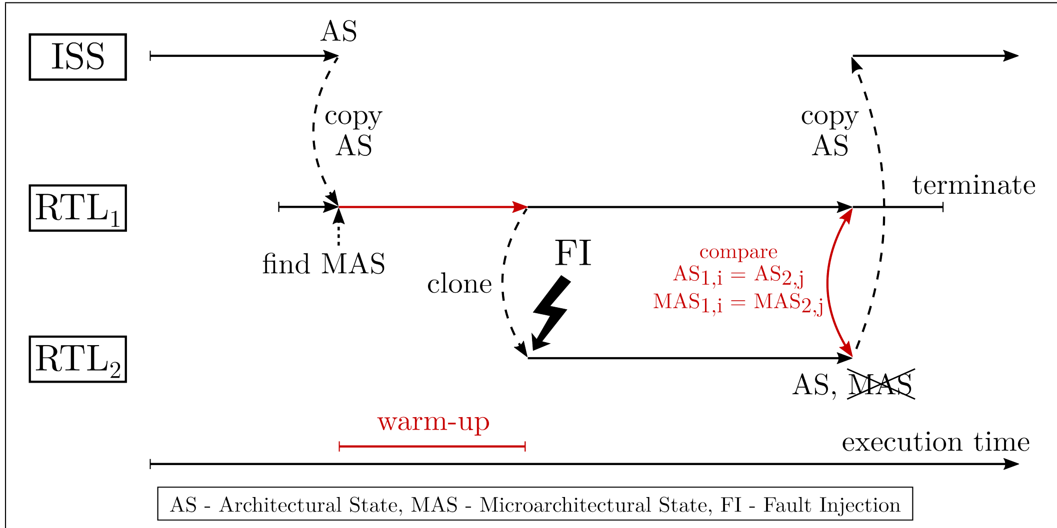


Figure 3.2.: Use of the runtime switch behavior in the fault injection scenario

3.2. Implementation Concept

In this section, the options for implementing a runtime switch behavior are discussed. On this basis, the best implementation approach is chosen, whose realization is described in the next chapter.

Three possible options for implementing the runtime switch behavior can be identified: Firstly, two instances of the FIVPs could be created and run side by side. This means that only few adjustments would have to be made to the VP itself and to the used CPU cores. However, the execution would have to be controlled from the outside, e.g., by a operating system multi-process environment. In case of a switch, the simulation of the first VP with the CPU at ISS level would be stopped and all states would be copied out of the simulation. These values would be copied into the second VP with the core at RTL and this VP would be started. The external manipulation would be very costly and possibly not feasible at all. Additionally, it is doubtful how performant this approach would be, since two SystemC kernels would be needed for two different VPs, which means that most components would simply be duplicated.

Contrary to the first option, the other two approaches only use one instance of the VP. The second idea is to swap the CPU or the core itself on a switch during simulation time. This would mean that when switching from ISS level to RTL, the states of the ISS would be saved and the CPU model would be removed from the VP. A CPU model with an RTL core would be instantiated and inserted in the same place. Next, the stored states would

be transferred to this core so that execution could then continue at RTL. The expected performance with this approach would certainly be higher than with the first option, since the overhead of duplicated simulation artifacts would be significantly lower. However, this option is technically difficult to implement. As described in Section 2.2, standard SystemC does not support the dynamic creation and binding of objects during runtime. For this reason, it should be impossible to remove a CPU model from the VP during runtime and connect another model to the same interface.

The third option is to incorporate multiple CPUs into one VP and develop a generic component that can activate and pause the individual CPUs. This controller would be a CPU multiplexer, so to speak, that could switch between different CPUs and forward the transactions of the respective active CPU to the remaining components of the VP. A CPU wrapping the ETISS core and a CPU with an RTL core could be connected to the CPU multiplexer. In addition, this approach can be easily extended if a second CPU with an RTL core is needed, e.g., as a non-faulty reference during fault injection. Furthermore, there is no need to instantiate and connect new components in SystemC during runtime, which is where the second approach failed. All components can be instantiated and connected during the elaboration phase of the SystemC kernel and the switch is exclusively controlled by the CPU multiplexer.

For the reasons mentioned above, the third approach will be implemented in this work. Figure 3.3 shows how the CPU multiplexer can be integrated into the VP and how it can be connected to different CPUs.
First, it should be noted that the mechanisms for activating and deactivating the individual CPUs are not shown in Figure 3.3. They are described in Section 4.1 and 4.2. What is shown, however, is that the CPU multiplexer provides two TLM sockets for each CPU, to which one instruction line and one data line is connected. As the CPU multiplexer itself is only connected to the system bus via two sockets, some form of demultiplexing is required. How the routing of the TLM transactions works and how it is implemented is discussed in Section 4.3. Likewise, it is noticeable that the interrupt lines are routed via the CPU multiplexer to the individual CPUs. The reason for this is explained in Section 4.5.

The last section in this chapter covers the development flow of the CPU multiplexer. Thereby the single steps, in which the development of the component is divided, are explained and the challenges of each step are emphasized.

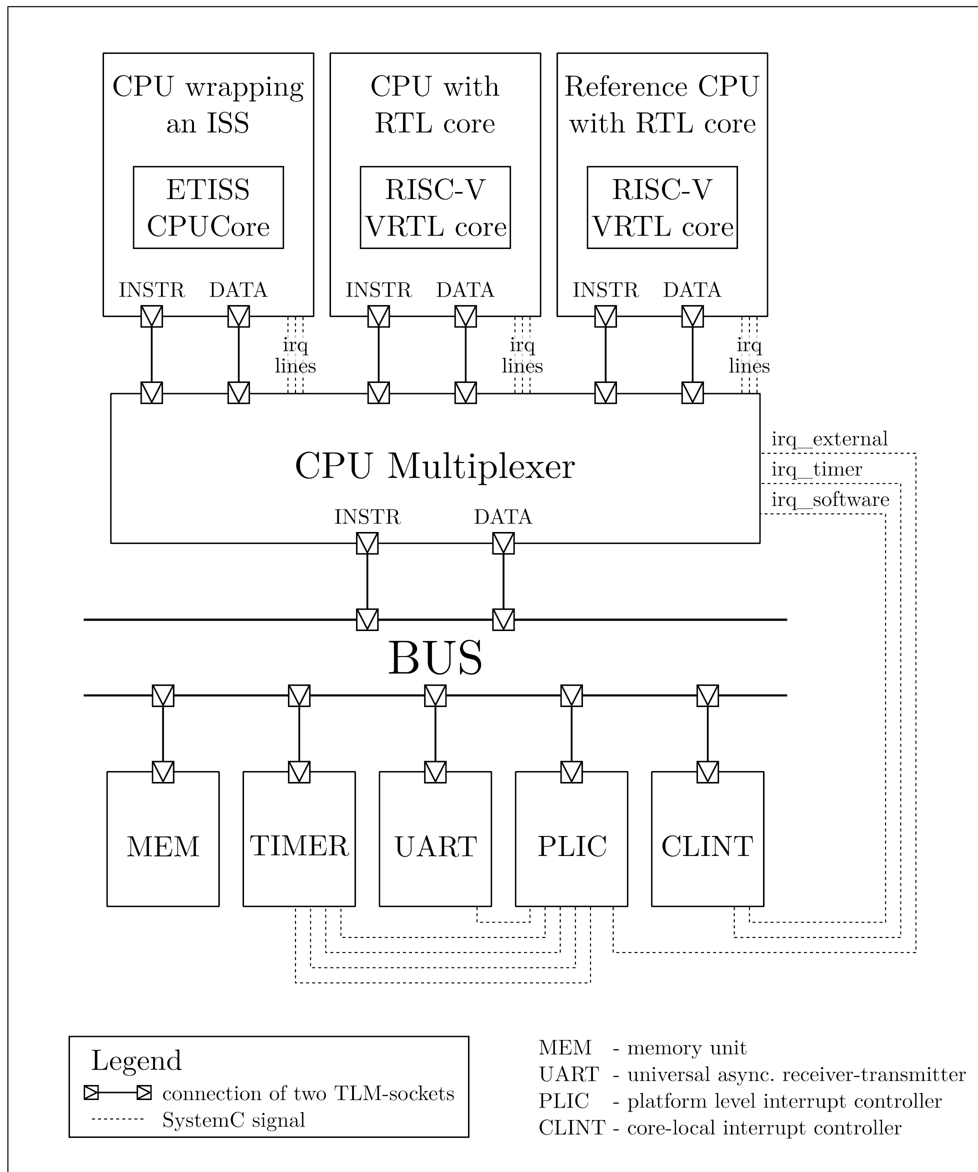


Figure 3.3.: Block diagram of the FIVP with the CPU multiplexer and multiple CPUs

3.3. Development Flow

When developing a runtime switch behavior, the question arises of how to verify the correctness of the switch and the jump-started core. To solve this problem, the second CPU with an RTL core can be used, which is later used as a reference CPU in the fault injection scenario. This is presented as RTL_1 in Figure 3.4. First, the CPU with the ISS and the reference CPU with the RTL core are started. Both run in parallel (tandem) until a certain point in time at which the switch takes place. Then, another CPU with an RTL core is being jump-started and the architectural and the micro-architectural states are set. Now, the reference CPU with the RTL core that has been running from the beginning can be used for comparison. Both RTL cores can be compared in each cycle and this way errors in the switch behavior can be found and the correctness of the switch can be verified.
However, this requires that the ISS and the reference RTL core were stopped on exactly the same instruction at the time of the switch. Likewise, the question arises how the ISS and the reference core can run side by side. Only one of them may commit to the memory, since they share the entire VP and memory operations may not be executed twice. For this reason, first a tandem simulation is developed that solves the described alignment problem and the memory problem. Only in the second step, a switch behavior is developed that can be verified using the procedure described above and shown in Figure 3.4.
The parallel simulation harms the performance, since in the actual sense the fast ISS is supposed to run on its own until the switch. Therefore, the reference kernel is removed as soon as the functionality of the switch behavior has been proved. It can be used later as a reference core for the fault injection but may then be jump-started as well.

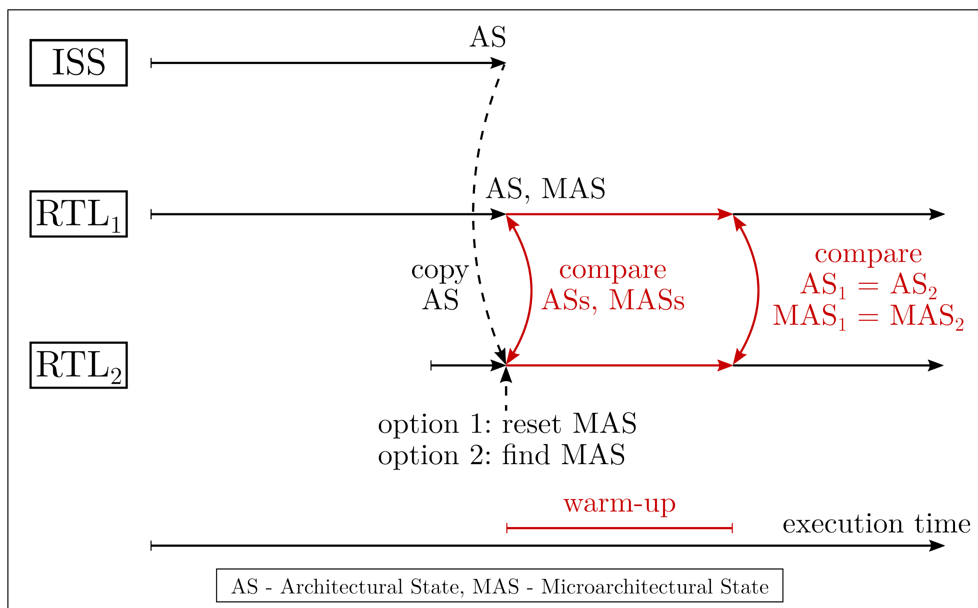


Figure 3.4.: Development approach for the switch from ISS level to RTL

After the tandem simulation and the switch behavior have been developed, a jump back to ISS level can be implemented. Since this should be rather trivial if the injected error has been masked and the RTL core still behaves correctly according to the ISA, the focus in this work will primarily be on the "switch down" shown in Figure 3.4.

4. Software Model of a CPU Multiplexer

This chapter describes the implementation and the exact functionality of the CPU multiplexer introduced in Section 3.2 and illustrated in Figure 3.3. The CPU multiplexer is a virtual prototype of a generic hardware component and is implemented in SystemC.

4.1. Clock and Reset Signal Forwarding

To enable all CPU models with different cores to be connected to the CPU multiplexer, they must all provide the same interfaces. For this reason all CPUs inherit from an abstract CPU base class. This class defines SystemC input ports for a reset and a clock signal, and two TLM initiator sockets. The platform-wide clock and reset signals are generated by signal generators and sent over wires to all components. Since it should be possible to execute and reset CPUs independently, these signals must be connected to the CPUs via the CPU multiplexer. The CPU multiplexer can then control the forwarding process and forward the clock and reset signals only to the currently active CPU(s). The CPU wrapping ETISS does not need the clock signal, but since the reset line is necessary, the clock signal is also forwarded to it for consistency. Possibly another CPU could be used later at this place, which needs the clock signal.

For forwarding the signals within the CPU multiplexer two `sc_methods` are defined: `forward_clk` and `forward_rst`. These are sensitive to the corresponding input ports and forward the signals according to the pseudo code in Listing A.1. The handling of the reset signals is analog to the sequence shown in Listing A.1. The forwarding function for the clock signal is also used to make some configurations in the first clock cycle. In addition, a cycle-based switch can be triggered from here by notifying a start-switch-on-this-cycle SystemC event.

4.2. Freezing and Waking up CPUs

As indicated above, it must be possible to activate and deactivate each CPU independently. For the CPUs with the RTL core this is very simple because they can be controlled via the clock signal. If a CPU is disconnected from the clock signal by the CPU multiplexer, it is deactivated, and if the clock signal is forwarded to it again, it continues to run. This is not so easy with the CPU that wraps ETISS. Since it does not use the clock signal, another way must be found to pause or stop the ISS. For this purpose, the function

`ISS_CPU::systemCallSyncTime` is useful, which ETISS calls at the end of each execution loop cycle (see Figure 2.5) to resynchronize with the SystemC time. In this function the check of a flag `freeze_cpu` is inserted and if the flag is set, a SystemC wait-function is called, which waits for a wake-up event. Thus, ETISS is deactivated or frozen so to speak until the event is notified. Via public functions the CPU multiplexer can set the freeze flag as well as notify the wake-up event.
This way two simple ways were found to activate and deactivate the CPUs with an RTL core and the CPU with the ISS.

4.3. TLM Routing for Memory Reads and Writes

The handling and demultiplexing of the instruction and data traffic within the CPU multiplexer is not as trivial as the forwarding of the clock and reset signals. As shown in Figure 4.1 the CPU multiplexer has two target sockets as interface for each CPU, one for the instruction traffic and one for the data traffic. For each of these six sockets, `b_transport` and `transport_dbg` functions must be implemented and registered. When a generic payload (GP) is sent to one of the sockets, the corresponding `b_transport` or `transport_dbg` function is called. This must then forward the GP to one of the two `tlm_initiator_sockets`, which can be accomplished with a simple function call. For the instruction line, this is easy because each CPU is allowed to read any number of instructions from the instruction memory and the reads do not compete. Therefore all three `tlm_target_sockets` of the CPU multiplexer to which the instruction read commands are sent may forward them directly to the instruction `tlm_initiator_socket`.

On the data path, however, this is not so simple. Since all three CPUs can run simultaneously and all execute the same instructions, they will all perform the same data read and write operations in the same order. However, the ISS will always run ahead because it completes one instruction per SystemC cycle. In contrast, for RTL cores it is the case that in certain cycles no instruction is completed because instructions might be speculatively fetched and decoded but not executed. This can be the case if a branch was not predicted correctly. Then, the pipeline is flushed and filled with new instructions which takes at least two cycles. In ETISS this stalling can not happen because it does not implement a pipeline on micro-architectural level. Therefore, it always runs ahead of the RTL cores.
For this reason, the read and write accesses via the TLM interfaces will arrive from all three CPUs at different times. It would be fatal if a write access to the memory would be allowed three times at different times. Likewise a data read access may not be permitted three times. For example, when reading from the UART interface, which is implemented using a receiver buffer register (RBR), the buffer register would be emptied and filled with a new data packet after each access. For this reason a way must be found to ensure that each memory read and write access can only be executed once in total. Since the ISS always runs ahead as stated above, it must wait for the other CPUs after each execution of a data read or write operation. As blocking transport functions are used in the simulation, this can be realized directly by a SystemC wait in the `b_transport` function. The data

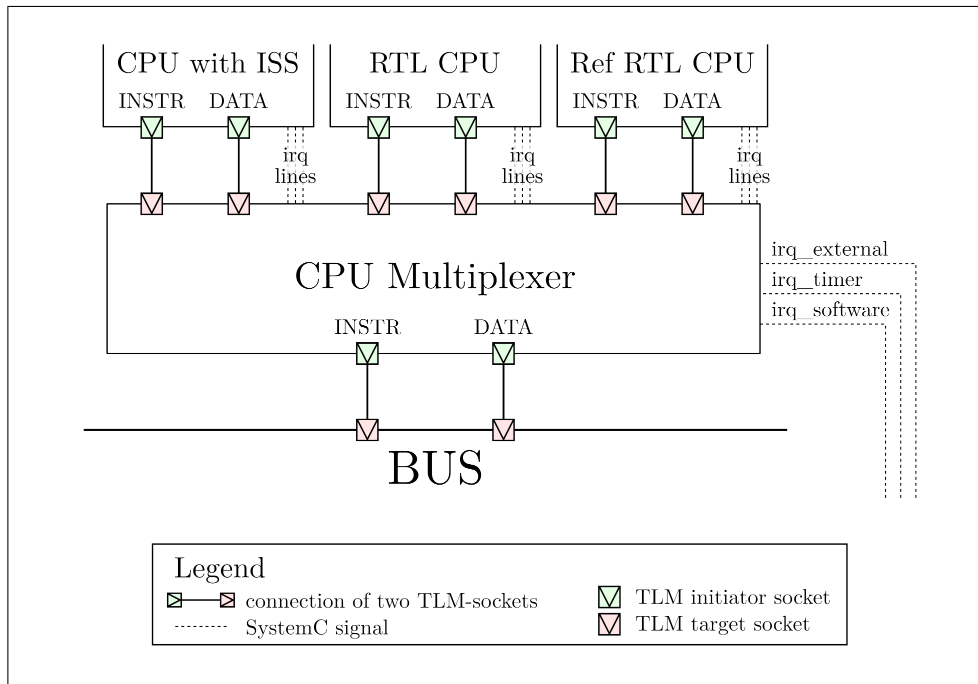


Figure 4.1.: CPU multiplexer and its TLM sockets

of the read or write access is stored in a buffer in the CPU multiplexer. As soon as the RTL CPU(s) want(s) to execute the same data read or write instruction, it is checked whether this is identical to the instruction stored in the buffer. If this is not the case, an error has occurred and the simulation must be terminated. If the data is identical, a write operation can return that it has already been executed and a read operation can return the result of the read access, which is also stored in the buffer. Afterwards the waiting CPU with the ISS can be woken up and all CPUs can continue their execution. If two CPUs with RTL cores are part of the simulation, it is always necessary to wait for the transaction of the last RTL core.

The entire process is illustrated in Figure 4.2. The procedure described here also has the advantage that CPUs running at the same time are always "approximately" synchronous, since they are synchronized on all read and write accesses. If only one CPU is run alone, there is of course no need for waiting and the TLM GPs can be forwarded immediately.

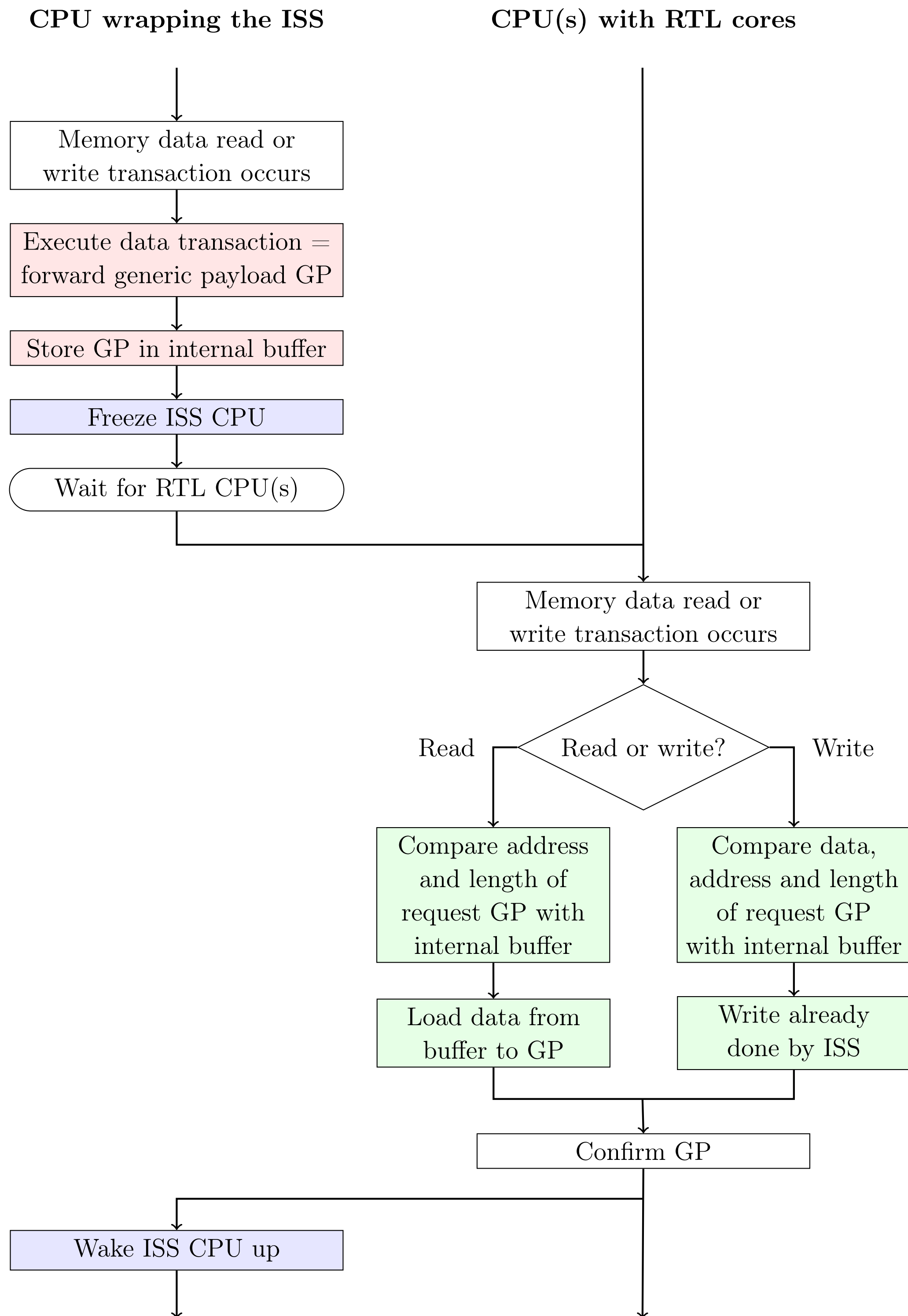


Figure 4.2.: Routing of TLM data transactions in the CPU multiplexer

4.4. Refinement Map

The refinement map contains all the code for the CPU multiplexer, which is processor specific, i.e., it cannot be used generally for every processor model. The main task of the refinement map is to define which architectural state in the ISS should be mapped to which state in the RTL model and how MAS should be treated.

To accomplish this, a so-called VRTLCoreStruct is defined in the refinement map. This structure is similar to the VirtualStruct in Etiss described in Section 2.5 and defines a list of registers and signals that are present in the RTL core. These include the instruction pointer (IP), general-purpose registers (GPRs), control and status registers (CSRs), and a few other control signals and registers. The VRTLCoreStruct stores both the references to the signals in the verilated RTL (VRTL) core and the name of the module, as well as the corresponding register in the ISS. In addition, the refinement map defines read and write functions for each register and signal that call direct programming interface (DPI) functions. These were described in Section 2.3 and can read or manipulate the registers and signals in the RTL core.

Lastly, the refinement map defines a function that can manually flush the RISC-V pipeline of the corresponding architecture. This is necessary for interrupt handling and switch functionality, which are described in Section 4.5 and 4.6. Listing 4.1 shows how the refinement map can be used to control the IP, GPRs and CSRs.

```
1   VRTLCoreStruct vrtl_core_;   // initialization not shown here
2
3   vrtl_core_.IP.read();
4   vrtl_core_.IP.write(0x80);
5
6   vrtl_core_.GPR.at(15).read();
7   vrtl_core_.GPR.at(15).write(0xd08ff);
8
9   for (auto &gpr : vrtl_core_.GPR)           // loop over GPRs
10  {
11      gpr.second.read();                     // works analogous
12      gpr.second.write(0x0);                 // for the CSRs
13  }
14
15  vrtl_core_.CSR.at(0x341).read();           // read MEPC
16  vrtl_core_.CSR.at(0x341).write(0xc0a);     // write MEPC
17
18  etissVirtualStruct->findName(vrtl_core_.IP.get_name())->read();
```

Listing 4.1: Example usage of the refinement map to manipulate the VRTL core

4.5. Interrupt Handling

As mentioned in Section 3.2 the CPU multiplexer must also intervene in the interrupt system. The reason for this is that when an interrupt occurs, an interrupt handler is called that first saves the GPRs to memory. If several CPUs are active at the same time this may lead to a problem because they share a common memory. To ensure that the GPRs from all CPUs have the same values, an alignment process must first be performed when an interrupt occurs. This means that the CPU with the ISS is frozen on an interrupt. Then, it is waited until all RTL CPUs have arrived at the same point. After that, the interrupt is forwarded to all CPUs and the CPU with the ISS is woken up so that they all start executing the interrupt handler at the same time.

The entire algorithm is shown in Figure 4.3 and implemented using an `sc_thread` that waits in an infinite loop for new interrupts. When an interrupt occurs, it is first determined on which interrupt line the interrupt has occurred. Afterwards, it is checked whether a switch process is currently being executed. If this is the case, it must be waited for its completion. Next, a flag is set to block all subsequent switch operations which is shown in the green block in Figure 4.3. It is not possible to execute a switch during the alignment for the interrupt forwarding. Thereafter, it is still necessary to wait until any memory access operations that may be taking place have been completed. During these the CPU with the ISS might be frozen.
If all previous conditions are fulfilled, the ISS CPU is frozen again. Meanwhile the interrupt is already forwarded to it. This allows a smoother wake-up process later. After that it is waited until the RTL CPU(s) have the same instruction in the ID stage that the ISS has stored in its machine exception program counter (MEPC) register. If this is the case, an additional check is made whether all GPRs of the RTL CPU(s) are equal to the GPRs of the ISS. If both is the case, all CPUs are aligned and the interrupt can also be forwarded to the RTL CPU(s). The problem with this is that the interrupt is not executed immediately by the RI5CY core after it has been raised, but only two cycles later. For this reason the pipeline must be flushed manually and the MEPC must be also written manually. Since the flush procedure highly depends on the implementation of the architecture, the implementation of the `flush_vrtl_pipeline_manually` function can be found in the refinement map described in the previous section.
If all this was successful the synchronous execution of the interrupt handler starts and it can be waited until it is finished. Finally, new interrupts can be waited for in the loop.

At this point it should be noted again that the whole complicated synchronization process is only necessary if several CPUs are active at the time of the occurrence of the interrupt. If only one CPU is active, the interrupt can of course be forwarded to it immediately.

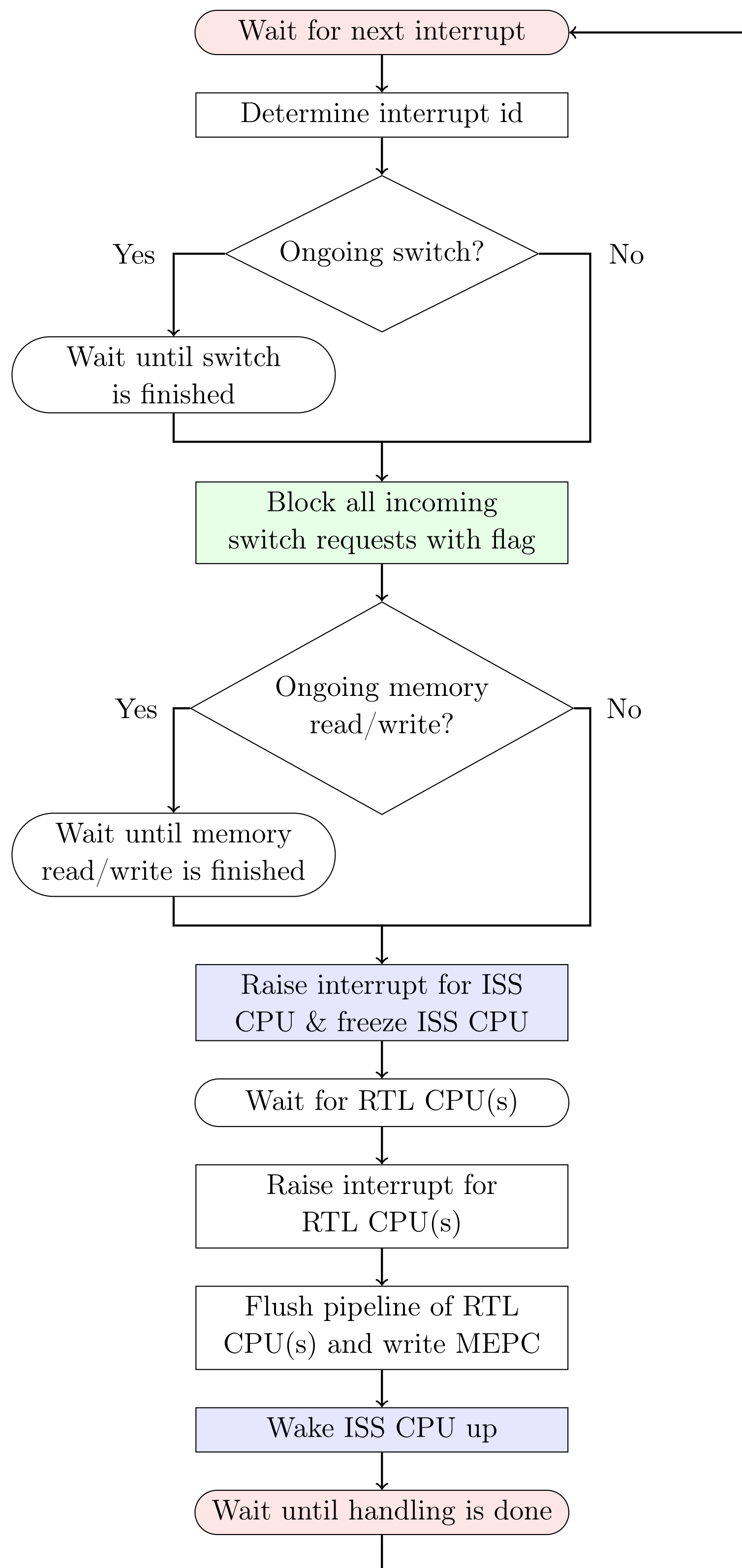


Figure 4.3.: Interrupt handling if several CPUs are active at the time of occurrence

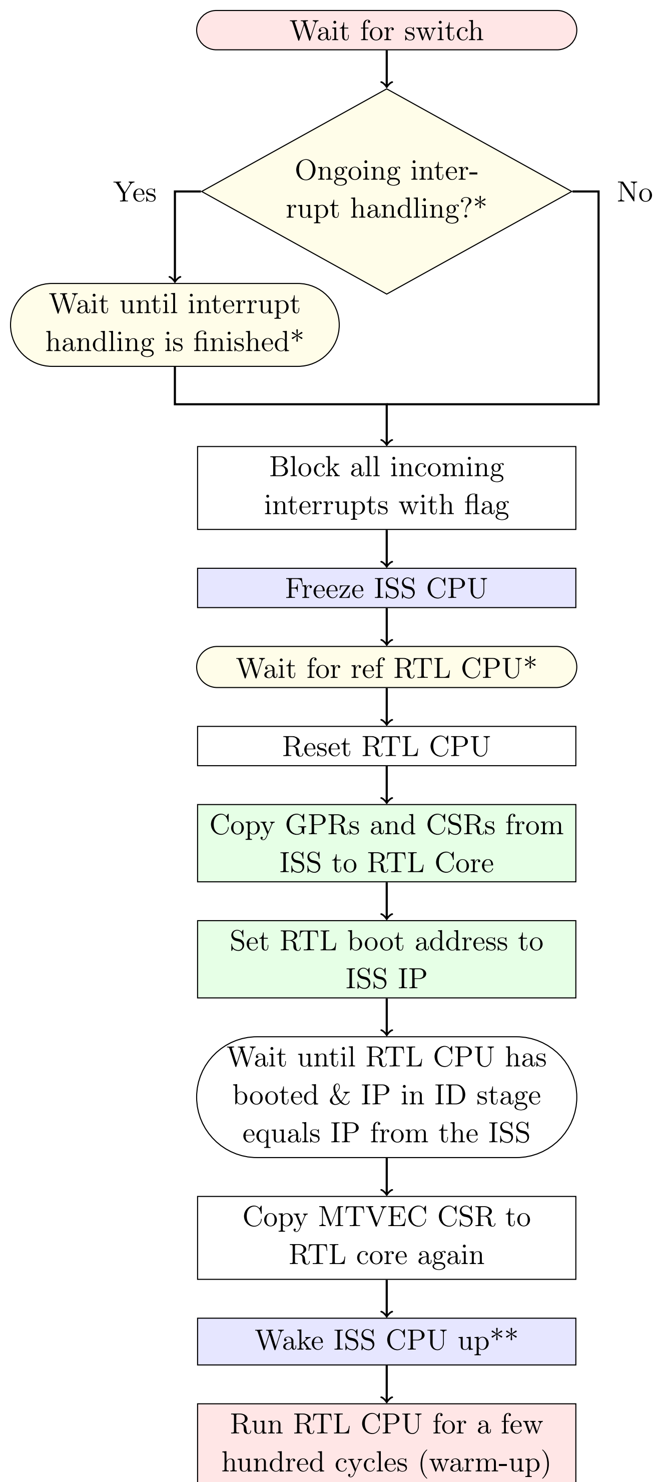
4.6. Switch Functionality

The execution of a switch in the CPU multiplexer works similar to the interrupt handling described in the previous section. In the following the single steps are described that are necessary for switching from the CPU on ISS level to the CPU on RTL. Thereby, the behavior developed in Chapter 3 is implemented. As already indicated there, the focus is first placed on the "switch down", i.e., on the switch from ISS to RTL, since this is the non-trivial switch. It is implemented analogously to Figure 3.4.

Just like the interrupt handling, the switch is implemented with an `sc_thread`. Its execution flow is shown in Figure 4.4. First, the `sc_thread` waits for an `sc_event` that triggers the switch. This can be notified, for example, in a certain cycle or depending on other events. Implementing this ensures that the switch can be triggered at any time. When a switch process is started, it is first checked whether an interrupt is currently forwarded within the CPU multiplexer, i.e., the procedure shown in Figure 4.3 is executed. This is only necessary if a reference RTL core is simulated in parallel to the ISS from the beginning, e.g., to verify the correctness of the switch. Therefore, the corresponding decision block in Figure 4.4 is highlighted in yellow. Thereupon, all incoming interrupts are blocked with the help of a flag. This is the flag used in Figure 4.3 to check if there is an ongoing switch. Blocking respectively postponing interrupts is not a problem, because the CPU wrapping the ISS is frozen in the next step. This means that the interrupt would have to wait until its wake-up anyway. The freezing and waking up of the ISS CPU is highlighted in blue in Figure 4.4. After freezing the ISS CPU, the reference RTL CPU must be waited for, if it is used in the above indicated way, and thereafter the jump-start of the CPU with the RTL core can begin. For this purpose the CPU is first reset.

After the reset all architectural states are copied from the ISS into the RTL core. The architectural states are the GPRs and the CSRs. At this point, other functions defined in the refinement map can also be called to initialize the micro-architectural states. However, this is not necessary for the RI5CY core used in this work. Afterwards, the boot address of the CPU with the RTL core can be set to the IP of the ISS and the boot process is started. When the IP arrives in the ID stage of the CPU, the machine trap-vector base address (MTVEC) CSR of the RTL core has to be set to the corresponding value in the ISS again, because it is reset during the boot process. After that, the CPU with the RTL core is ready. If a tandem simulation is to be executed or the CPU wrapping the ISS is needed again later, e.g., because a return jump is to be executed, it can now be woken up. From this point on, all interrupts can be processed again. This completes the switch process.

After a few hundred warm-up cycles, in which all previously unused micro-architectural states can be set, the RTL core can be cloned into the reference RTL core and, e.g., fault injection can be performed.



* Only necessary if reference RTL CPU is used to verify the switch
** If no jump back is required the ISS CPU wake up can be omitted

Figure 4.4.: Procedure of a switch from the CPU with ISS to the CPU with RTL core

5. Experimental Evaluation - Results

5.1. Evaluation Approach

After the development and implementation of the runtime switch behavior in the form of the CPU multiplexer, this is now to be tested and evaluated experimentally. The developed algorithm and its implementation must be examined and evaluated in two dimensions. The first dimension is the correctness of the functionality. It must be verified that the switch from ISS level to RTL is always executed correctly and that the CPU with the RTL core behaves afterwards as if it had been running from the simulation start. In the second dimension, the performance and speed-up of the simulation must be evaluated.

To evaluate the functionality and the performance of the CPU multiplexer and the FIVP in which it has been integrated, benchmark programs are required. In the context of this work, two test scenario programs are used, which are briefly described in the following.

- **FreeRTOS**: The first test used is a FreeRTOS based test program. FreeRTOS is an open-source real-time operating system that is especially used on microcontrollers and small microprocessors [22]. It comes with a lean kernel that includes a scheduler to execute various tasks. The test program used here first initializes and boots the FreeRTOS kernel and then executes the main task. In this task a delay function is called three times with a delay of one millisecond. When the timer interrupt occurs after one millisecond, a message is printed on the UART. After three ticks, i.e., three simulated milliseconds, the program execution and thus the simulation is terminated.
- **Dhrystone**: The second test program used is the classic Dhrystone benchmark. The latest version used here is version 2.1 in the programming language C and was published in 1988 by Reinhold P. Weicker [23]. The first version was also developed by Weicker in 1984 in the Ada language [24]. Dhrystone is an easy to use integer benchmark, which is until today one of the standard benchmarks for CPU and compiler performance measurements especially for microprocessors. It was developed as an integer-based counterpart to Whetstone, which is also a single-program benchmark, but based on floating-point arithmetic. For the examination of the CPU multiplexer 10000 Dhrystone runs are executed.

In order to evaluate the correctness of the functionality of the switch process, both benchmark programs are used. With the FreeRTOS program the functionality of the switch behavior can be examined before, during and after interrupts. With the Dhrystone

program the functionality can be examined during arithmetic, load, and store operations of the CPU and it can be ensured that in the long run no errors occur. After all, 10000 Dhrystone runs in an RTL core take over 3.5 million clock cycles. Of course, the switch functionality cannot be tested on each of these 3.5 million clock cycles. Therefore, a test script was run that performs 200 random switches per benchmark program. After the switch, the program was simulated to the end in parallel with Extendable Translating Instruction Set Simulator (ETISS) and the reference RTL core, and the states of all CPUs were compared regularly.

To evaluate the performance of the CPU multiplexer and the FIVP, the Dhrystone program is to be used because it was developed exactly for this purpose. On the one hand, this evaluation is important because many new SystemC components and function calls have been added to the simulation. It must be ensured that this overhead does not slow down the simulation so much that the acceleration of the simulation, which is supposed to be achieved by the switch concept, is not compensated. On the other hand, the speed-up achieved and the potential of the runtime switch behavior should also be described quantitatively.
For this purpose, the following five measurement series with the FIVP and built-in CPU multiplexer and three reference measurement series with the FIVP without the CPU multiplexer are conducted:

1. **Solo ETISS simulation**, with CPU multiplexer, no tracing
2. **Solo VRTL simulation**, with CPU multiplexer, no tracing
3. **Solo VRTLmod[1] simulation**, with CPU multiplexer, no tracing
4. **Tandem simulation** of ETISS and one VRTLmod[1] core, no tracing
5. **Triple simulation** of ETISS, one VRTLmod[1] core and another reference VRTLmod[1] core, no tracing

6. **Reference ETISS simulation**, without CPU multiplexer, no tracing
7. **Reference VRTL simulation**, without CPU multiplexer, no tracing
8. **Reference VRTLmod[1] simulation**, without CPU multiplexer, no tracing

As indicated above, in all measurement series 10000 Dhrystone benchmark runs are executed and the performance is measured in executed clock cycles per second (CPS). The measurements are performed on the *Intel Xeon Gold 6126 CPU @ 2,60 GHz* and the system memory is 287 GiB. The detailed simulation results can be found in the Appendix. In the next section, the average performance results for all eight measurement series are presented and discussed.

[1]The VRTLmod core is a modified version of the verilated RTL (VRTL) core. The modification includes the Fault Injection API. Therefore, this core is generally slower than the unmodified VRTL core.

5.2. Results

The procedure described in the previous section was successfully used to prove the functionality of the runtime switch behavior. At all 400 randomly selected switch points, the switch was successful and the simulation subsequently completed without errors. To the outside, the CPU with the jump-started RTL core and the CPU with the reference RTL core, which ran from the beginning of the simulation, behaved the same at all times. Additionally, the comparison with the architectural states of the reference RTL core was always successful.

To evaluate the performance, the eight scenarios defined in the previous section were simulated. The average performance values of all measurements with the CPU multiplexer are shown in the following table.

1. ETISS	**2. VRTL**	**3. VRTLmod**	**4. Tandem**	**5. Triple**
$1.030 \cdot 10^6$ CPS	$138.3 \cdot 10^3$ CPS	$130.2 \cdot 10^3$ CPS	$54.6 \cdot 10^3$ CPS	$26.5 \cdot 10^3$ CPS

Table 5.1.: Simulation results average values (see Appendix for all results)

The simulation with the ISS is naturally the fastest. It is faster by a factor of eight than the simulation with the modified VRTL (VRTLmod) core. This is obviously due to the different detail levels, as already introduced in Chapter 1. The tandem simulation of ETISS and a VRTLmod core is 60% slower than the simulation with a VRTLmod core and the triple simulation is even 80% slower. This can be explained by the fact that the ISS has to wait for the RTL core(s) at every memory transaction. Also, the cores and the ISS are not really simulated in parallel, but it must always be jumped back and forth. This is very expensive and it can be stated that in no case the tandem or triple simulation should be carried out from start to finish.

Next, the performance with the CPU multiplexer shall be compared to the reference measurements without the CPU multiplexer. For this purpose, the measured average performance values of the last three scenarios are listed below.

6. ETISS Reference	**7. VRTL Reference**	**8. VRTLmod Reference**
$1.274 \cdot 10^6$ CPS	$242.9 \cdot 10^3$ CPS	$210.8 \cdot 10^3$ CPS

Table 5.2.: Reference simulation results average values (see Appendix for all results)

Comparing the values from Table 5.2 with Table 5.1, it is noticeable that the performance is generally decreased by the implementation of the CPU multiplexer. This has already been described in the previous section and can be explained by the fact that the CPU multiplexer adds many software modules and function calls per simulated cycle and thus reduces the overall performance of the simulation. Table 5.3 quantifies this performance

decrease. It can be seen that the CPU multiplexer slows down the simulation with ETISS by about 19% and the simulation on RTL by about 40%.

ETISS ⑥ → ①	**VRTL** ⑦ → ②	**VRTLmod** ⑧ → ③
$\frac{1.03 - 1.274}{1.274} = -19.2\,\%$	$\frac{138.3 - 242.9}{242.9} = -43.1\,\%$	$\frac{130.2 - 210.8}{210.8} = -38.2\,\%$

Table 5.3.: Solo simulation performance decrease due to CPU multiplexer

Finally, it must be verified and demonstrated that the application of the runtime switch behavior yields a performance gain that far exceeds the performance decrease shown in Table 5.3. Of course, the performance gain of the runtime switch behavior without switching back to the ISS strongly depends on the length of the test program and the switch point. Therefore, the overall performance of the simulation with switching is to be examined here in dependence of the switching point. For this, it is assumed that a test program of one million cycles is used and that a period of 10k cycles is to be examined after the switch. This period includes the warm-up phase of the RTL cores after the switch. The period length of 10k cycles is chosen based on [11] where a period of 10k cycles is used as a reference in experimental evaluation. In this work it is assumed that during the period of 10 cycles ETISS, a jump-started RTL core and a jump-started reference RTL core are simulated. This is equivalent to the triple simulation that is Scenario 5 in the previous section. After the 10k cycles ETISS and the reference core can be stopped and the simulation is finished on RTL. The overall performance can be calculated with the following formula. For $\mathrm{PRF_{ETISS}}$, $\mathrm{PRF_{TRIPLE}}$ and $\mathrm{PRF_{VRTLMOD}}$ the corresponding performance values from Table 5.1 are inserted.

$$PRF_{TOTAL} = \frac{x \cdot PRF_{ETISS} + 10^4 \cdot PRF_{TRIPLE} + (10^6 - 10^4 - x) \cdot PRF_{VRTLMOD}}{10^6}$$

The above function is plotted in blue in Figure 5.1. In addition, the performance of the simulation with ETISS is plotted in green and with VRTLmod in red. The blue graph shows that the overall performance for the case of a switch without switch back increases the later the switch is performed. This is due to the fact that a larger proportion is then simulated using the fast ISS. Figure 5.1 also shows that the simulation with the switch outperforms the reference VRTLmod simulation as soon as the switch point lies beyond the 100000th cycle. If it is before, the reference VRTLmod simulation is faster. For this reason, a switch back to the ISS should be implemented in the future. Then one would not have to simulate with the VRTLmod core until the end and could use $\mathrm{PRF_{ETISS}}$ instead of $\mathrm{PRF_{VRTLMOD}}$ in the last term of the formula above. This would make the middle term negligibly small and the overall performance would be constant at about $\mathrm{PRF_{ETISS}}$. This is the green graph in Figure 5.1. Compared to the simulation with the reference VRTLmod core, this could result in a performance gain of about $\frac{1030-210.8}{210.8} = 389\,\%$.

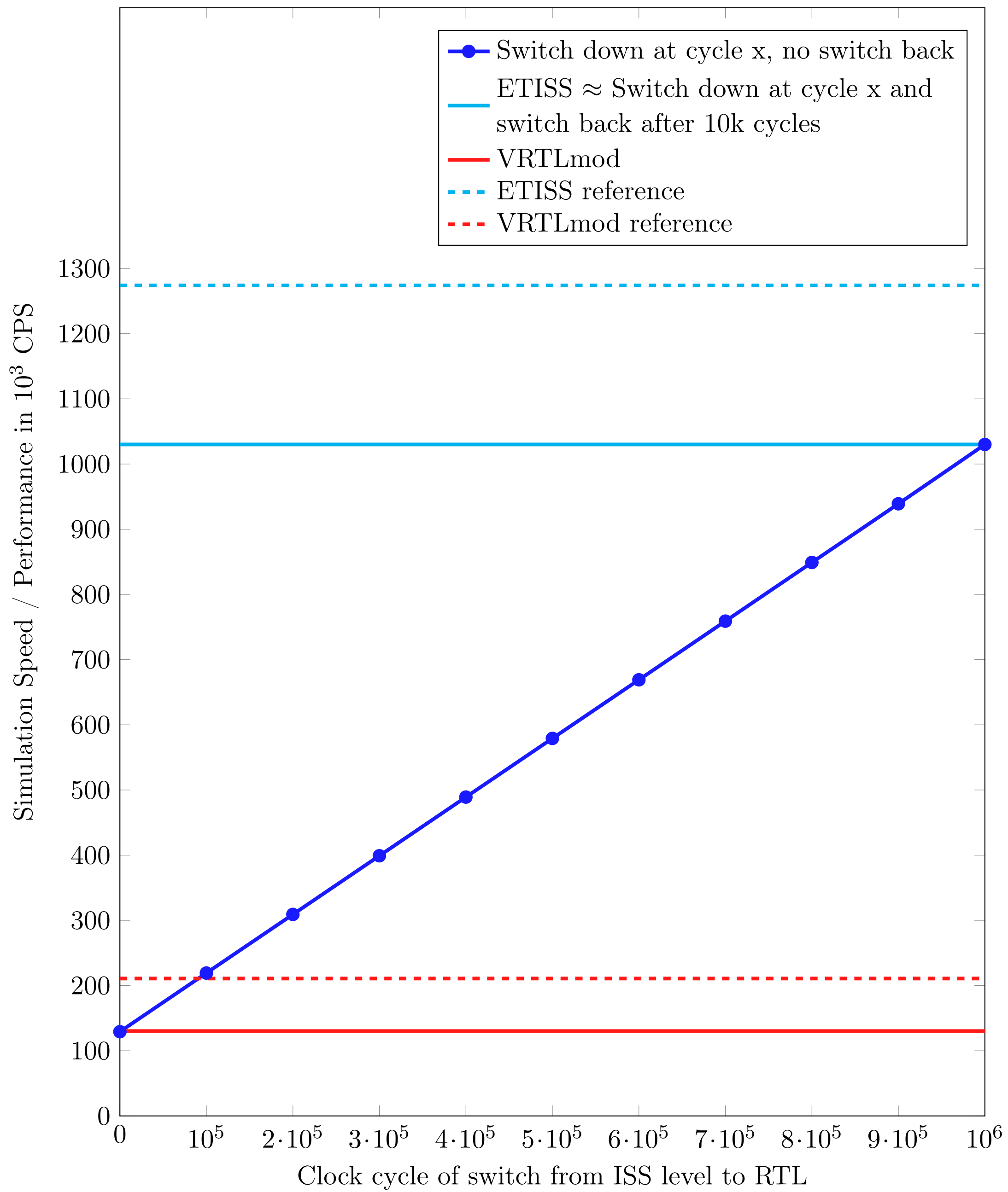


Figure 5.1.: Performance estimation for different switch cycles with and without switch back, a fixed warm-up and cool-down length of 10k cycles and a test program of 1M cycles

6. Summary and Outlook

In this work, a runtime switch behavior for virtual platforms (VPs) was developed that makes it possible to switch between different CPU models during the simulation runtime. These CPU models can be implemented at different levels of detail, so that it is possible to switch from a high performance CPU model, simply wrapping an instruction set simulator (ISS), to a slower CPU containing a core on register transfer level (RTL) that is in turn much more detailed.

First, the runtime switch behavior was developed conceptually and three ways were shown to implement it. It turned out that the option of incorporating all CPUs into one VP and then developing a component that can switch between the CPUs was the most feasible approach. For this reason, the virtual prototype of a CPU multiplexer was developed, which, as the name implies, can switch between different CPU models. To do this, it can activate and deactivate the CPUs individually and independently, control the data flow to the system bus and manage the interrupt lines. On the one hand the CPU multiplexer allows to run all CPUs alone, to perform a tandem simulation of one ISS and one RTL core or to execute a triple simulation with an additional reference RTL core. On the other hand, it also allows to switch from the CPU with the ISS to a CPU with an RTL core.

Within the scope of this work, all functionalities of the CPU multiplexer were evaluated experimentally with respect to their functionality and performance. From the functionality perspective, the CPU multiplexer and the runtime switch behavior implemented therein work flawlessly. Although the installation of the CPU multiplexer in a VP reduces the performance compared to the reference simulation with a single CPU by up to 43 %, the simulation with a switch from ISS to RTL outperforms the reference simulation by a multiple. However, it was also shown that the overall performance strongly depends on the switch time point, since it is not yet possible to switch back to ISS level as soon as the detailed simulation of the CPU is no longer required.

Nevertheless, in order to achieve the maximum simulation performance, a switch back should be implemented in the future. Therefore an algorithm must be found, which can determine, when a switch back is appropriate and possible. The switch process itself can then be implemented very easily, because the CPU multiplexer provides all necessary functionalities. The switch back from the RTL to the ISS level is the more trivial switch, since the ISS is a more abstract model and all the states that the ISS implements are simulated in the RTL model.

A. Appendix

A.1. Clock Signal Forwarding

```
void CpuMultiplexer::forward_clk()
{
   case SimulationMode::etiss:
      forward_clk_to_iss_cpu();
   case SimulationMode::vrtl:
      forward_clk_to_vrtl_cpu();
   case SimulationMode::tandem:
      forward_clk_to_vrtl_cpu();
      forward_clk_to_iss_cpu();
   case SimulationMode::switch:
      forward_clk_to_iss_cpu();
      if not SwitchStatus::switching
         forward_clk_to_ref_vrtl_cpu();
      if SwitchStatus::switching
      or SwitchStatus::vrtl_active
         forward_clk_to_vrtl_cpu();
}
```

Listing A.1: Pseudo code of the sc_method that forwards the clock signal

A.2. Simulation Results: Performance Measurement for the FIVP with CPU Multiplexer

In each of the five measurement series, 10000 Dhrystone benchmark runs are executed and the performance is measured in CPS. The measurements are performed on the following processor: Intel Xeon Gold 6126 CPU @ 2,60 GHz. The system memory is 287 GiB.

Index	1	2	3	4	5	6	7	8	9	10	Avg
Performance [MCPS]	1.027	1.032	1.034	1.026	1.020	1.026	1.027	1.035	1.030	1.040	1.030

Table A.1.: Results Scenario 1 (CPU multiplexer, solo **ETISS**, no tracing)

Index	1	2	3	4	5	6	7	8	9	10	Avg
Performance [kCPS]	136.1	~~112.3~~	140.0	139.3	141.2	134.9	137.9	137.6	139.3	~~122.5~~	138.3

Table A.2.: Results Scenario 2 (CPU multiplexer, solo **VRTL**, no tracing)

Index	1	2	3	4	5	6	7	8	9	10	Avg
Performance [kCPS]	123.5	131.3	129.5	130.8	130.5	131.4	130.3	130.4	133.5	130.5	130.2

Table A.3.: Results Scenario 3 (CPU multiplexer, solo **VRTLmod**, no tracing)

Index	1	2	3	4	5	6	7	8	9	10	Avg
Performance [kCPS]	54.5	55.4	54.3	55.5	55.0	54.0	53.9	55.8	53.2	54.8	54.6

Table A.4.: Results Scenario 4 (CPU multiplexer, tandem **ETISS-VRTLmod**, no tracing)

Index	1	2	3	4	5	6	7	8	9	10	Avg
Performance [kCPS]	26.3	26.5	26.5	26.6	26.6	26.5	26.5	26.6	26.5	26.5	26.5

Table A.5.: Results Scenario 5 (CPU multiplexer, triple **ETISS-VRTLmod**, no tracing)

A.3. Simulation Results: Reference Performance Measurement for the FIVP without CPU Multiplexer

In each of the three measurement series, 10000 Dhrystone benchmark runs are executed and the performance is measured in CPS. The measurements are performed on the following processor: Intel Xeon Gold 6126 CPU @ 2,60 GHz. The system memory is 287 GiB.

Index	1	2	3	4	5	6	7	8	9	10	Avg
Performance [MCPS]	1.277	1.276	1.282	1.291	1.239	1.271	1.272	1.283	1.275	1.277	1.274

Table A.6.: Results Scenario 6 (reference, no CPU multiplexer, **ETISS**, no tracing)

Index	1	2	3	4	5	6	7	8	9	10	Avg
Performance [kCPS]	241.0	246.8	238.3	241.3	246.4	241.4	241.6	249.9	243.5	238.7	242.9

Table A.7.: Results Scenario 7 (reference, no CPU multiplexer, **VRTL**, no tracing)

Index	1	2	3	4	5	6	7	8	9	10	Avg
Performance [kCPS]	208.3	213.6	208.5	208.7	~~158.8~~	211.7	212.2	~~155.5~~	209.9	213.7	210.8

Table A.8.: Results Scenario 8 (reference, no CPU multiplexer, **VRTLmod**, no tracing)

Acronyms

ALU **A**rithmetic **L**ogic **U**nit. 17, 18

API **A**pplication **P**rogramming **I**nterface. 13, 38

AS **A**rchitectural **S**tate. 22, 35

CLINT **C**ore-**l**ocal **Int**errupt Controller. 19

CPS Clock **C**ycles **p**er **S**econd. 38, 39, 41, 44, 45

CPU **C**entral **P**rocessing **U**nit. iii, 1–3, 13, 15, 16, 19–26, 28–40, 42, 49, 50

CSR **C**ontrol and **S**tatus **R**egister. 16–18, 21, 32, 35

DMI **D**irect **M**emory **I**nterface. 11

DPI **D**irect **P**rogramming **I**nterface. 13, 14, 32

EDA **E**lectronic **D**esign **A**utomation. 2, 19

ELF **E**xecutable and **L**inkable **F**ormat. 19

ETISS **E**xtendable **T**ranslating **I**nstruction **S**et **S**imulator. iii, 2–5, 15, 16, 20, 21, 24, 28, 29, 38–41, 44, 45, 49, 50

EX **Ex**ecution Stage. 18

FIVP **F**ault **I**njection **V**irtual **P**latform. 2, 3, 5–7, 19–21, 23, 25, 37, 38, 49

FPR **F**loating-**p**oint **R**egister. 17, 19

FPU **F**loating-**p**oint **U**nit. 18, 19

GCC **G**NU **C**ompiler **C**ollection. 15

GDB **G**NU **D**e**b**ugger. 16

GP **G**eneric **P**ayload. 12, 29, 30

GPR **G**eneral-**p**urpose **R**egister. 13, 16, 17, 19, 21, 32, 33, 35, 50

HDL **H**ardware **D**escription **L**anguage. 13, 14

ID **I**nstruction **D**ecode Stage. 18, 19, 33, 35

IEEE **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers (professional association). 8, 13

IF **I**nstruction **F**etch Stage. 18

IP **I**nstruction **P**ointer. 13, 15, 16, 32, 35

ISA **I**nstruction **S**et **A**rchitecture. 2–5, 15–17, 27

ISS **I**nstruction **S**et **S**imulator. iii, 2–5, 15, 20–23, 26–30, 32, 33, 35–37, 39–42, 49

JIT **J**ust-**i**n-**t**ime Compiler. 15, 16

KIPS **K**ilo **I**nstructions **p**er **S**econd. 21

LLVM **L**ow **L**evel **V**irtual **M**achine. 15

LSU **L**oad-**S**tore **U**nit. 18

MAS **M**icro-**a**rchitectural **S**tate. 22, 32, 35

MEM **Mem**ory. 19

MEPC **M**achine **E**xception **P**rogram **C**ounter. 33

MIPS **M**illion **I**nstructions **p**er **S**econd. 1, 21

MTVEC **M**achine **T**rap-**Vec**tor Base Address Register. 35

OS **O**perating **S**ystem. 11

PLIC **P**latform **L**evel **I**nterrupt **C**ontroller. 19, 20

PULP **P**arallel **U**ltra **L**ow **P**ower Projekt. 18

RISC **R**educed **I**nstruction **S**et **C**omputer. 17

RTL **R**egister **T**ransfer **L**evel. iii, 2–5, 13, 20–24, 26–30, 32, 33, 35–42, 49

SoC **S**ystem **o**n **C**hip. iii, 1, 2, 6, 19

TCC **T**iny **C** **C**ompiler. 15

TLM **T**ransaction-**L**evel **M**odeling. 8, 10–12, 19, 20, 24, 28–31, 49

TUM **T**echnical **U**niversity of **M**unich. 2, 15, 19

UART **U**niversal **A**synchronous **R**eceiver-**T**ransmitter. 19, 29

VP **V**irtual **P**latform (system of Virtual Prototypes). 1–4, 6–8, 11, 19–21, 23, 24, 26, 42

VRTL **V**erilated **R**egister **T**ransfer **L**evel Model. iii, 14, 20, 21, 32, 38–40, 44, 45, 50, 51

VRTLmod **Mod**ified **V**erilated **R**egister **T**ransfer **L**evel Model. 38–41, 44, 45, 50

WB **W**rite**b**ack Stage. 18

List of Figures

1.1. Multi-level simulation flow suggested by Mueller-Gritschneder [11] 5
1.2. Difference between ETISS-ML and the idea of this work 5

2.1. Phases of a SystemC simulation 9
2.2. Transaction between a TLM initiator socket and a target socket 12
2.3. Verilator flow 14
2.4. Etiss architecture [17] 15
2.5. Etiss core execution loop [18] 16
2.6. Block diagram of the RI5CY processor [19] 18
2.7. Block diagram of the FIVP 20

3.1. General idea of a runtime switch behavior 22
3.2. Use of the runtime switch behavior in the fault injection scenario 23
3.3. Block diagram of the FIVP with the CPU multiplexer and multiple CPUs 25
3.4. Development approach for the switch from ISS level to RTL 26

4.1. CPU multiplexer and its TLM sockets 30
4.2. Routing of TLM data transactions in the CPU multiplexer 31
4.3. Interrupt handling if several CPUs are active at the time of occurrence . . 34
4.4. Procedure of a switch from the CPU with ISS to the CPU with RTL core 36

5.1. Performance estimation for different switch cycles with and without switch back 41

List of Tables

2.1. RISC-V integer register file with 32 GPRs [20] 19

3.1. Properties and advantages of different simulation levels 21

5.1. Simulation results average values 39
5.2. Reference simulation results average values 39
5.3. Solo simulation performance decrease due to CPU multiplexer 40

A.1. Results Scenario 1 (CPU multiplexer, solo ETISS) 44
A.2. Results Scenario 2 (CPU multiplexer, solo VRTL) 44
A.3. Results Scenario 3 (CPU multiplexer, solo VRTLmod) 44
A.4. Results Scenario 4 (CPU multiplexer, tandem ETISS-VRTLmod) 44
A.5. Results Scenario 5 (CPU multiplexer, triple ETISS-VRTLmod) 44
A.6. Results Scenario 6 (reference, no CPU multiplexer, ETISS) 45
A.7. Results Scenario 7 (reference, no CPU multiplexer, VRTL) 45
A.8. Results Scenario 8 (reference, no CPU multiplexer, VRTLmod) 45

List of Listings

4.1. Example usage of the refinement map to manipulate the VRTL core . . . 32

A.1. Pseudo code of the sc_method that forwards the clock signal 43

Bibliography

[1] H. Herold, B. Lurz, and J. Wohlrab, *Grundlagen der Informatik*, 2nd ed. Munich, Germany: Pearson Deutschland GmbH, 2012.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach*, 6th ed. Cambridge, MA, United States: Morgan Kaufmann, 2019.

[3] TSMC, *28nm Technology*. [Online]. Available: `https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_28nm` (visited on 09/18/2022).

[4] J. Rossignol, *Apple's First 3nm Chips for MacBook Pro Expected to Enter Production This Year*, Aug. 22, 2022. [Online]. Available: `https://www.macrumors.com/2022/08/22/3nm-chip-production-for-upcoming-macs-report/` (visited on 09/18/2022).

[5] IBM, *IBM Unveils World's First 2 Nanometer Chip Technology*, May 6, 2021. [Online]. Available: `https://newsroom.ibm.com/2021-05-06-IBM-Unveils-Worlds-First-2-Nanometer-Chip-Technology,-Opening-a-New-Frontier-for-Semiconductors` (visited on 09/18/2022).

[6] ESA, *Virtual Platform Technology*. [Online]. Available: `https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/Virtual_Platform_Technology` (visited on 09/21/2022).

[7] J. Geier, "Fast RTL-based Fault Injection Framework for RISC-V Cores", M.S. thesis, TU Munich, Mar. 6, 2020.

[8] Y. Xing, A. Gupta, and S. Malik, "Generalizing Tandem Simulation: Connecting High-level and RTL Simulation Models", in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 154–159. DOI: `10.1109/ASP-DAC52403.2022.9712564`.

[9] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification", *ACM Transactions on Design Automation of Electronic Systems*, vol. 24, 2018. DOI: `10.1145/3282444`.

[10] D. Aarno and J. Engblom, *Software and System Development using Virtual Platforms*, 1st ed. Waltham, MA, United States: Morgan Kaufmann, 2015.

[11] D. Mueller-Gritschneder, M. Dittrich, J. Weinzierl, E. Cheng, S. Mitra, and U. Schlichtmann, "ETISS-ML: A Multi-Level Instruction Set Simulator with RTL-level Fault Injection Support for the Evaluation of Cross-Layer Resiliency Techniques", in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 609–612. DOI: 10.23919/DATE.2018.8342081.

[12] T. de Schutter, *Better Software. Faster!*, 1st ed. Mountain View, CA, United States: Synopsys Inc., 2014.

[13] IEEE, "IEEE Standard for Standard SystemC Language Reference Manual", *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012. DOI: 10.1109/IEEESTD.2012.6134619.

[14] IEEE, "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language", *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018. DOI: 10.1109/IEEESTD.2018.8299595.

[15] W. Snyder, *Verilator Manual*, Jul. 2022. [Online]. Available: https://www.veripool.org/ftp/verilator_doc.pdf (visited on 09/26/2022).

[16] Veripool, *Welcome to Verilator*. [Online]. Available: https://www.veripool.org/verilator/ (visited on 09/26/2022).

[17] TUM, Chair of Electronic Design Automation (EDA), *ETISS (Extendable Translating Instruction Set Simulator)*. [Online]. Available: https://github.com/tum-ei-eda/etiss (visited on 09/27/2022).

[18] TUM, Chair of Electronic Design Automation (EDA), *ETISS 0.8.0*. [Online]. Available: https://tum-ei-eda.github.io/etiss/index.html (visited on 09/27/2022).

[19] A. Traber, M. Gautschi, and P. D. Schiavone, *RI5CY: User Manual*, ETH Zurich, Switzerland and University of Bologna, Italy, Apr. 2019. [Online]. Available: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf (visited on 09/29/2022).

[20] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual*, May 2017. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (visited on 09/26/2022).

[21] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design", in *Proceedings of the 50th Annual Design Automation Conference*, Austin, Texas, 2013. DOI: 10.1145/2463209.2488859.

[22] Amazon Web Services, Inc., *The FreeRTOS™ Kernel*. [Online]. Available: https://www.freertos.org/RTOS.html (visited on 10/07/2022).

[23] R. P. Weicker, "Dhrystone benchmark: Rationale for version 2 and measurement rules", *SIGPLAN Not.*, pp. 49–62, Aug. 1988. DOI: 10.1145/47907.47911.

[24] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark", *Commun. ACM*, pp. 1013–1030, Oct. 1984. DOI: 10.1145/358274.358283.