A Flexible Uncore Infrastructure for RISC-V Core Development

Michael Jungmair
Technical University of Munich
Garching near Munich, Germany
michael.jungmair@tum.de

Tobias Schmidt
Technical University of Munich
Garching near Munich, Germany
to.schmidt@tum.de

Alexis Engelke
Technical University of Munich
Garching near Munich, Germany
alexis.engelke@tum.de

Armin Ettenhofer
Technical University of Munich
Garching near Munich, Germany
armin.ettenhofer@tum.de

Felix Krayer
Technical University of Munich
Garching near Munich, Germany
felix.krayer@tum.de

Jonas Lauer
Technical University of Munich
Garching near Munich, Germany
jonas.lauer@tum.de

Malte von Ehren
Technical University of Munich
Garching near Munich, Germany
malte.von.ehren@tum.de

Martin Schulz
Technical University of Munich
Garching near Munich, Germany
schulzm@in.tum.de

Abstract

The openness, flexibility, and modularity of RISC-V opens the door for new developments in computer architecture research. The same characteristics also make it highly attractive for teaching and computer architecture education, as they lower the entry barrier for students in the implementation of their own core designs, especially in combination with FPGA-based rapid prototyping. However, one major issue remains: the development of the surrounding environment, including but not limited to caches, the memory consistency model as well as debugging and control protocols, remain complex and often detract from the ability to design the cores themselves.

To address this challenge, we propose a flexible uncore infrastructure — targeted at FPGAs — that can be used to implement, test and use the full range of RISC-V core design possible. Our uncore infrastructure contains a configurable cache and memory hierarchy for multi-core support matching the RISC-V memory semantics and features an easily adaptable communication protocol to simplify the collection of performance metrics and debugging information. We show that our approach allows for the development of both performance-oriented cores as well as cache-coherent multi-core systems, while only requiring minimal resources on the targeted FPGA. It enables core designers to focus on the essential part of their design, which makes the framework attractive for researchers and students alike. We use this framework successfully in freshman bachelor courses with positive results.

Keywords: Processor Development, Multi-Core Systems, RISC-V, FPGA, VHDL

1 Introduction

The RISC-V ISA [8] is a rapidly evolving, open, and flexible architecture suitable for developing new and possibly more specialized architecture extensions. Based on these properties, it is not only a good match for innovative research, but also for teaching and education. For the latter, in particular, the explicitly narrow scope and size of the base ISA enables the development of new, yet fully functional, processor cores with comparably low effort, even compared to other RISC freely usable architectures.

Both research and teaching with RISC-V are frequently conducted on FPGAs, as they offer a good trade-off between easy-to-implement simulations and high performance. However, the development of processor cores on FPGAs comes with a rather high entry barrier: the required uncore logic to connect, feed, test and use a newly designed core is non-trivial and requires significant work. This includes proper configurations of the FPGA attached DDR memory, the connection to I/O devices (which is often hardware-specific), the need for a multi-core capable cache hierarchy with cache coherency, and the support of the matching memory consistency model; all hardware-specific and non-trivial issues that need to be addressed for each new core design project. Further, each core design project needs to communicate with the FPGA host to enable easy debugging and performance analysis. As a consequence, a significant amount of engineering overhead must be invested, detracting from the actual research or education targets.

To address this issue, we propose a flexible and modular *uncore* infrastructure to simplify the VHDL-based development of multi-core RISC-V processors from scratch on FPGAs. Our infrastructure provides a cache hierarchy to support multiple cores, atomic memory operations, and a generic interface for accessing memory and I/O components.

To account for comparably low memory latency resulting from the lower clock rate of the FPGA itself, the access latency to the caches and the main memory is configurable. Additionally, an easily extensible communication protocol captures debugging information, performance statistics, and other data from all infrastructure components.

We show that this infrastructure allows for both the development of complex cores with various performance optimizations and the development of simple cores for the emulation of many-core systems. Additionally, our approach simplifies a direct and fair comparison of different processor core implementations and allows for evaluating application workloads in different environments. Our experiences in teaching show a significantly lower entry barrier for freshman students when developing RISC-V cores, allowing them to implement much more optimized core designs in a shorter time frame.

The main contributions of this paper are the following:

- An uncore infrastructure for simplified development of RISC-V cores for computer architecture research and teaching, supporting multiple processor cores.
- A simple and yet flexible interface between a multi-core processor infrastructure and the actual core implementation, allowing for the development of small and simple as well as more complex cores.
- Two case studies with different cores showing what is possible within the proposed uncore environment.
- An evaluation showing the low overhead of our uncore infrastructure in terms of needed FPGA resources.

The remainder of this paper is structured as follows: in Section 2, we describe our uncore infrastructure, and in Section 3, we give more details on our communication protocol. Then, in Sections 4 and 5, we show two case studies illustrating the flexibility of our approach with regard to processor core implementations and many-core systems. In Section 6, we report development experiences using this infrastructure. Finally, in Section 7, we cover related work, and in Section 8, we summarize our findings.

2 Approach for Flexible Uncore

An infrastructure for the development of processor architectures suited for research and teaching demands a design that is both easy to understand and extend. In particular, such an infrastructure must fulfill the following requirements:

- Ease of development for both simple and more complex, performance-oriented RISC-V cores.
- Support for advanced features, like instruction buffers or atomic instructions for multi-core systems.
- Accurate simulation of memory latency and memory access patterns.
- Support for multiple memory and I/O devices on varying hardware platforms.

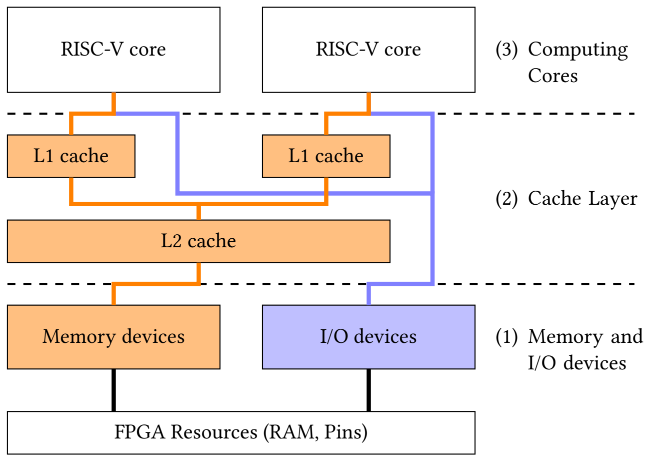

Figure 1. Uncore Infrastructure

To achieve a flexible design matching these requirements, we split the processor infrastructure into three layers (c.f. Figure 1): (1) a layer for memory and I/O devices, (2) an optional cache layer, and (3) the interface to the processor cores themselves. We use a bus-based communication protocol between the three layers, allowing us to replace each layer independently of the others. This design facilitates the development of new processor cores and the evaluation of different memory hierarchies with multi-core support. Furthermore, we can migrate the system quickly to other hardware platforms or emulate it on standard CPUs. All components are implemented in VHDL and new projects include the files to make use of our uncore infrastructure. In the following, we provide a brief overview of the four main components: the three layers and the bus protocol connecting them.

2.1 Memory and I/O Layer

The memory and I/O layer provides a unified interface for accessing the FPGA's hardware components, including the onboard DDR3 memory. Additionally, hardware clocks, software interrupt registers and communication buffers can be implemented as memory-mapped I/O devices. The layer's implementation depends on hardware-specifics, like clock frequency or memory capacity. We, therefore, designed it to be thin and easily retargetable to the respective FPGA hardware platform, while all other layers remain FPGA hardware agnostic. Moreover, this design facilitates the simulation of the system with open source tools, like GHDL [4].

All available memory and I/O devices are connected with the next higher layer to receive read and write operations. We assign disjoint address ranges to each device and use the most significant bit in the address to distinguish between memory and I/O devices. As FPGAs, unlike real processors, operate typically with only a few hundred MHz, the latency of memory operations can be increased by the user to emulate realistic memory access times in terms of the number of CPU cycles needed.

2.2 Cache Layer

We added the second layer to support more advanced processor designs with a cache hierarchy and multi-core support. We provide a two-level write-back cache hierarchy that implements a directory-based MESI protocol for cache coherency [5] and atomic memory operations for multi-core synchronization. Like many modern processors, we use 512-bit large cache lines and implement four-way set-associative caches on both levels with an least-recently-used eviction policy. As shown in Figure 1, the first level cache (L1) is a private cache for both instruction and data and corresponds to the L2 cache in today's Intel [6] or AMD [1] processors. We implemented the atomic memory operations from the "A" standard extension of the RISC-V ISA [8] in the L1 cache. The cores in the next higher layer trigger these instructions by appending the atomic operation's opcode to the memory operations. The L1 caches are connected through a shared bus to our last-level cache (L2). Since accessing memory-mapped I/O devices triggers side effects, we forward these operations directly to the underlying layer. Every cache also collects in-depth statistics on cache hits/misses, evictions, invalidations and fetches. As with the memory devices, the cache access latencies can be changed to reach realistic access timings.

2.3 Interface for Computing Cores

On the computing core layer, the actual RISC-V cores are implemented. The cores can perform writes, reads, or atomic memory operations on 8-byte aligned memory words; such requests are sent to the cache layer below. The core can also specify an additional mask that disables individual bytes to implement unaligned memory operations or accesses with less than eight bytes.

In addition, it is also possible to load an entire cache line, which can be used for further optimizations inside the core, for example, implementing an instruction buffer. However, we currently do not propagate the signal used for cache coherency to the layer of the computing cores. Hence, the executed program has to flush the instruction buffer manually using the `FENCE.I` instruction.

2.4 Bus Protocol

For the communication between the components in our infrastructure, we designed a generic request-acknowledgment bus protocol. Figure 2 shows the transmission between a requesting and a responding party in detail. The requesting party sets the enable signal and encodes in the payload the operation to perform and the memory address. The responding party enables the acknowledgment to signal that the result is available. The operation completes as soon as the respondent device resets the acknowledgment signal. We use

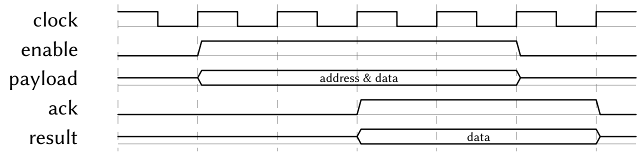

Figure 2. Bus protocol

this protocol in various places: to connect the three layers or to communicate between the L1 and L2 cache. The signal flow is identical in all instances of this protocol, but depending on the use case, the size of the payload may change or additional information such as opcodes for atomic memory operations are encoded.

3 Debugging and Communication Protocol

For debugging and evaluating our infrastructure as well as the RISC-V cores, we need a stable connection between the host system and the system under evaluation, synthesized on an FPGA or running as simulation. While there are many standards for connecting two systems that offer high performance, e.g., PCI or Ethernet, for our purposes, portability is more important than achieving high throughput or low latencies. Thus, we rely on UART-based serial interfaces for physical transport: it is available on almost all FPGA development boards, can be easily emulated during simulation, and is supported by a wide range of software libraries.

On top of a serial byte-wise interface, we use a package-based protocol that is generic enough for all of our requirements: as we do not want to embed test programs into our bitstreams, we need to boot the system from outside. This includes resetting the running system, loading programs into main memory, and enabling program execution. Additionally, we want to enable stepping through program execution, i.e., stop program execution and resume it later. Furthermore, we need the possibility to introspect the system's state from outside: this includes CPU registers and additional custom registers for debugging as well as performance counters to enable detailed benchmarking, e.g., from the cache hierarchy. In addition to supporting the boot process, enable debugging and detailed benchmarking, we also require a communication channel with complex software running on the RISC-V cores via virtual I/O devices that send and receive data via the same protocol used for booting and debugging.

Based upon these requirements, we design a requester-responder protocol where the host system acts as the requester and sends requests via a serial interface. These requests are first buffered and then distributed to the responders on a central shared bus. Every component inside our infrastructure with a relevant state (e.g., memory controller, the individual caches, CPU) listens to all requests on this bus until it detects a request with a matching component

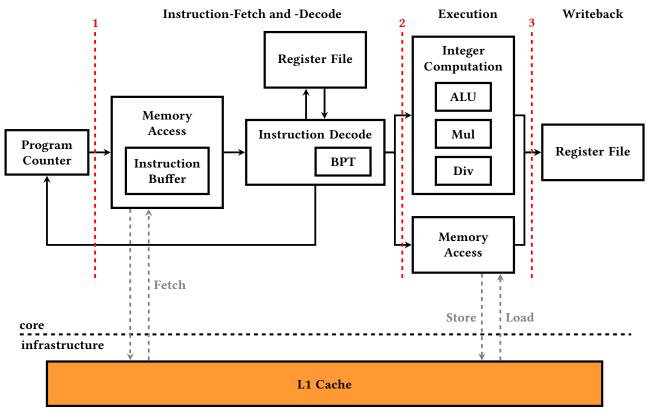

Figure 3. Structure of the pipelined processor with in-order execution.

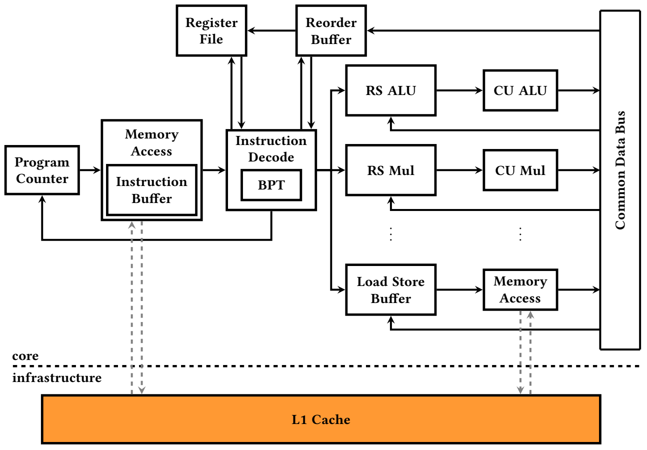

Figure 4. Structure of the pipelined processor with out-of-order execution (pipeline stages are not shown here).

identifier. It then processes the request depending on the contained function identifier (e.g., read register, write register, write memory) and sends a reply via the bus. This reply is then checked and forwarded to the serial interface. This distributed request processing design allows each component to perform arbitrarily complex actions based on a request, such as directly interacting with the memory controller. However, most components only process very simple requests, namely reading and writing (control-)registers. To facilitate this, we provide an embeddable VHDL component that handles such limited requests transparently.

4 Case Study 1: Performance-oriented Cores

As the first case study, we show that our infrastructure can be used to develop performance-focused cores.

4.1 Core Description

We use our uncore infrastructure to implement, test and compare two different RISC-V cores, which implement RV64IM [8] and can execute arbitrary RV64IM code. The current implementation, however, is limited to the unprivileged specification.

The first core uses a simple pipeline scheme with in-order execution, illustrated in Figure 3. Experimenting with timing measures on the FPGA allows us to reduce the number of pipeline steps to the three depicted in the figure. Thus, in combination with a single-cycle Branch Prediction Table (BPT), we can reduce the number of cycles lost to a branch misprediction to just one, without needing to flush, making the implementation even simpler. Furthermore, we implement an instruction buffer that loads chunks of 16 instructions from the L1-Cache, ensuring fast instruction fetching.

The second core is a single-issue out-of-order core implementing the Tomasulo algorithm [10]. As seen in Figure 4, it consists of a register file, a reorder buffer to keep track of register availability and flushing, computation units (CU), and reservation stations (RS) of varying sizes to handle the parallel execution of different types of instructions as well as a common data bus to bundle results. Although most instructions can be completely reordered, all load and store instructions are executed in-order with respect to each other to avoid the complexity of memory hazard detection. Some components like the memory access and instruction buffer are shared with the first core with minimal adjustments. Since the computation units for division and multiplication are themselves pipelined, they can start one operation every cycle. Therefore, long executions do not block computation units when flushing. The number of cycles needed for flushing depends logarithmically on the size of reorder buffer.

4.2 Integration with Uncore Infrastructure

Both core implementations adhere to the shared interface of the uncore infrastructure, which attaches them to the cache hierarchy. To improve execution performance, the cores use the infrastructure's capability to access whole cache lines to fill the instruction buffer.

The UART communication described in Section 3 is utilized to extract more details about the exact state of the components. This data currently includes utilization of the reservation stations and the reorder buffer, the data currently on the common data bus, as well as in- and outputs of the computation units, but can be easily extended to specific requirements. Figure 5 shows an example of how valuable this data can be, where we measure the utilization of the reservation stations for arithmetic/logic operations during the execution of different programs on a Xilinx VC707 FPGA. Thus, we can use the interface to make a detailed analysis with different numbers of reservation stations and reorder buffer sizes to find an optimal configuration.

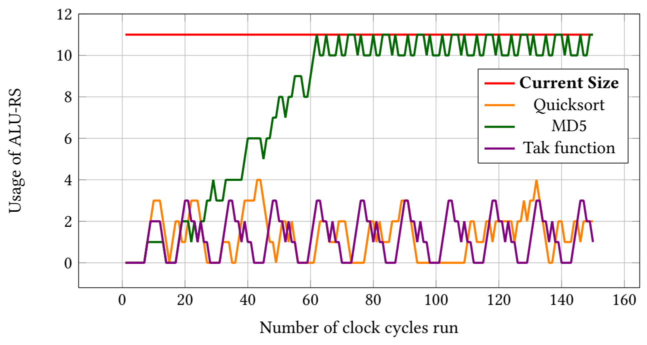

Figure 5. Utilization of the ALU reservation stations

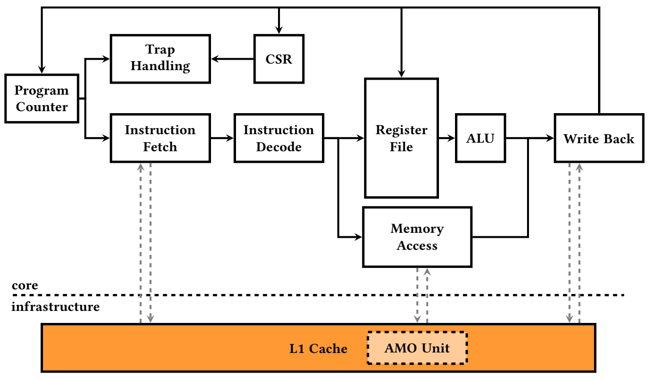

Figure 6. Structure of the size-optimized processor core with basic support for parallel programming

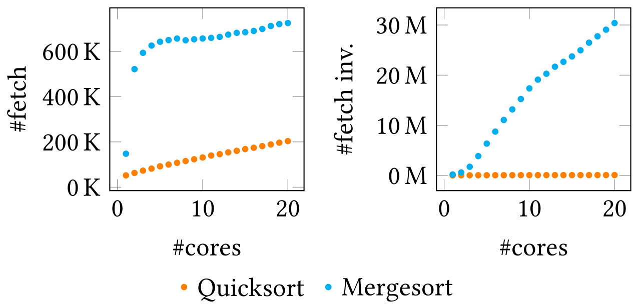

Figure 7. L2-Cache statistics for fetches (left side) and invalidating fetches (right side)

In addition to performance metrics, we also rely on the communication infrastructure to extract debugging information and controlling the core during development, like accessing the program counter, register values and operating the core in single-step execution. To further allow bidirectional communication with the running program, we use a blocking memory-mapped I/O-component to implement read and write buffers.

5 Case Study 2: Multi-core Systems

In our second case study, we implement a parallel RISC-V processor with multiple cores.

5.1 Core Description

To achieve high concurrency, we use small processor cores, which, in addition to RV32IM, also support most of the specified atomic instructions of RV32A [8]. Combined with coherent caches, this also enables synchronization. To facilitate multi-core programming even further, we also implement several instructions from the privileged instruction set [9] to support, e.g., unique hart IDs and interrupt handlers. To keep the RISC-V cores small enough to fit many of them on a single FPGA, we implement individually scalar cores, as shown in Figure 6, without further optimizations like pipelining or out-of-order execution, although given sufficient space on FPGAs more complex cores could also be added in such a multi-core design.

5.2 Integration with Uncore Infrastructure

Being able to configure and build a parallel system with many cores allows users to analyze the properties of parallel programs for varying settings, including different cache configurations. For example, consider the two statistics extracted from the L2 caches in Figure 7: it shows the number of fetches and fetch-invalidates for parallel implementations of quicksort and mergesort. Fetches occur when a core reads from a missing cache line; a fetch-invalidate is caused by writing write on a missing cache line. As quicksort works in-place, the number of fetches and the number of fetch-invalidates is low, as only few synchronisation is required. However, since mergesort allocates memory frequently, more synchronization is required. This causes cache-stealing, i.e., processors contend with each other for cache lines, which causes many expensive fetch-invalidates.

6 Experiences

In the following, we describe our experiences with our uncore infrastructure for developing RISC-V-based systems with regard to technical aspects, but also in the context of the core development process itself.

Technical Aspects. Based on our case studies, our infrastructure provides the foundation for developing new RISC-V cores in single-core and multi-core environments from scratch. It allows for the development of both simple/small cores, as well as more complex and performance-oriented RISC-V core designs. The infrastructure design generally imposes few limitations on the core design itself, and the possibility to perform memory accesses of up to 512 bits allows for further optimizations inside the core. The main remaining limitation for the development of high-performance cores is the missing possibility of integrating caches deeply into the core, as it requires modifying the core interface. However, this adaption is possible with comparably low effort.

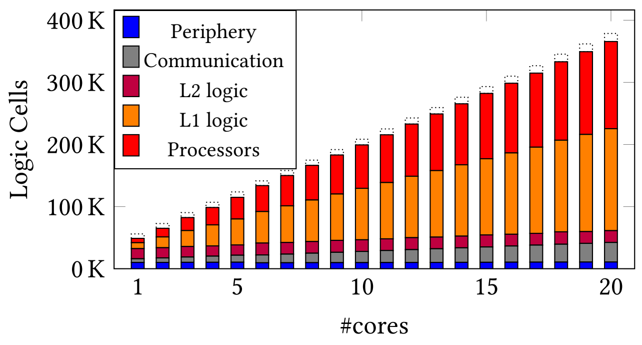

Figure 8. Resource Utilization for different core counts. Note that the data in L1 and L2 caches is not stored in logic cells but special memory not included in this figure.

The infrastructure also scales well with an increasing number of cores: with a simple core design, we can fit 20 working cores on our target board, a Xilinx VC707 board with 485k logic cells. The distribution of logic cells to the core and other components of our infrastructure is shown in Figure 8. Note that a full system with one core uses only ~70k cells, therefore being small enough to fit on smaller development boards, and that the size used by components other than the cores themselves and the L1 cache is usually negligable, especially for an increasing number of cores. With even larger boards, our current design allows up to 64 cores.

Core Development. During the development of cores, two key features of our infrastructure significantly reduce the effort of the development process: first, the flexible debugging possibilities enable extracting many kinds of data from different components of the core. And second, this allows for a fast simulation of the core on an FPGA with extensive data collection at the same time. Further, the possibility to simulate the entire infrastructure in software reduces the overhead of potentially high development times.

Finally, when analyzing and optimizing core designs, the ability to collect detailed performance statistics of various components is an additional benefit.

Usage in Teaching. Based on its property, we also successfully use this infrastructure in a teaching environment. In a freshman undergraduate lab course, small groups of students are given the assignment to implement a RISC-V core from scratch on an FPGA.

In previous years without this uncore infrastructure, students spent a significant amount of time in properly configuring hardware and I/O devices. Generally, this allowed students to implement simple functional, but unoptimized cores in a time frame of a semester.

In recent years, when we provided our infrastructure as foundation, students were able to implement much more sophisticated core designs, often including pipelining and other optimizations, even in a shorter time frame, allowing them to cover more further aspects in class.

7 Related Work

A recent work providing an infrastructure and an interface for a user's core implementation is the "Bring Your Own Core" (BYOC) framework [2]. Like our infrastructure, it can be executed on an FPGA, but rather focuses on providing support for multiple cores with different ISAs running simultaneously on the system. As common for high-performance cores, every core must implement its own L1 cache, which in turn has to implement the protocol of the L2 cache given by BYOC. Since we want to encourage the integration of quickly developed as well as high-performance cores, we provide an L1 cache, but do also plan to support cores with their own caches.

A prominent example of a RISC-V processor is SonicBOOM [12]. Similar to our infrastructure, it can be synthesized to run on an FPGA, but uses cache optimizations currently not present in our caches, like even-odd banking, a next-line prefetcher and a line fill buffer. Due to the high integration between the processor core and its environment, it is more difficult to use a different core within the SonicBOOM for testing purposes, which limits its ability to act as a general uncore infrastructure.

Several projects aim to teach students the development of pipelined [7] or out-of-order [11] cores, also providing them with an infrastructure to test their implementations. However, these infrastructures currently do not support multiple cores or provide protocols for communication with I/O devices when executed on hardware. More insight into the processor behavior during execution on an FPGA is achieved by Bulić et al.'s "FPGA-based integrated environment" [3] that students can use to run and debug assembly programs, while the processor core is not intended to be changed.

8 Summary

In this paper, we described a flexible uncore infrastructure for the development of VHDL-based multi-core RISC-V processors. We described our approach for this infrastructure consisting of three layers, which interface with memory and I/O devices on one end, with a cache hierarchy for coherent support of multiple cores in the middle, and with pluggable computing cores at the other end. Additionally, the infrastructure provides a versatile communication interface for gathering performance statistics and other debugging information. Our experiences show that this infrastructure allows for the development of performance-oriented RISC-V cores as well as scalable multi-core systems. In addition to using it in RISC-V-based computer architecture research, this infrastructure can be used for teaching purposes as well and has been successfully tested at TUM for this purpose.

References

[1] Advanced Micro Devices, Inc. 2021. *AMD64 Architecture Programmer's Manual.*

[2] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M Nguyen, Yaosheng Fu, Florian Zaruba, et al. 2020. BYOC: a" bring your own core" framework for heterogeneous-ISA research. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 699–714.

[3] Patricio Bulić, Veselko Guštin, Damjan Šonc, and Andrej Štrancar. 2013. An FPGA-based integrated environment for computer architecture. *Computer Applications in Engineering Education* 21, 1 (2013), 26–35.

[4] Tristan Gingold et al. 2021. GHDL. https://github.com/ghdl/ghdl, accessed 2021-05-12.

[5] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[6] Intel Corporation. 2021. *Intel 64 and IA-32 Architectures Software Developer's Manual.*

[7] Jason Lowe-Power and Christopher Nitta. 2019. The Davis In-Order (DINO) CPU: A Teaching-focused RISC-V CPU Design. In *Proceedings of the Workshop on Computer Architecture Education*. 1–8.

[8] RISC-V Foundation. 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified.*

[9] RISC-V Foundation. 2019. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, DocumentVersion 20190608-Priv-MSU-Ratified.*

[10] R. M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33. https://doi.org/10.1147/rd.111.0025

[11] Stephen A Zekany, Jielun Tan, James A Connelly, and Ronald G Dreslinski. 2021. RISC-V Reward: Building Out-of-Order Processors in a Computer Architecture Design Course with an Open-Source ISA. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 1096–1102.

[12] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*.