Towards a Rocket Chip Based Implementation of the RISC-V GPC Architecture

Lars Luchterhandt[a], Tom Nellius[a], Robert Beck[a], Rainer Dömer[b], Pascal Kneuper[a], Wolfgang Mueller[a], and Babak Sadiye[a]
[a]Paderborn University/Heinz Nixdorf Institute, Paderborn, Germany
Email: {larluc,beckr,tnellius}@mail.upb.de, {pkneuper,wmueller,babaks}@hni.upb.de
[b]University of California, Irvine, USA
Email: doemer@uci.edu

Abstract

RISC-V has received worldwide acceptance in the industry and by the academic community. As of today, multiple RISC-V applications and variants are under investigation for embedded IoT systems, from resource-limited single-core processors up to multi-core systems for High-Performance Computing (HPC). Recently, the Grid of Processing Cells (GPC) platform has been proposed as a scalable parallel grid-oriented network of processor cores with local memories. This paper describes a prototype design of the GPC platform for hardware implementation at Register-Transfer Level (RTL) based on modified RISC-V Rocket processors with scratchpad memories. It introduces a scalable Chisel-based implementation of the modified Rocket cores with RTL generation and a functional test using Verilator simulation. This work also includes the adaptation of the Chipyard software toolchain to extend the compiler to multi-core grids with different local address spaces.

1 Introduction

The open standard RISC-V instruction set architecture (ISA) has received worldwide acceptance as a viable option besides ARM-based systems. As of today, many implementations and variants are under investigation covering multiple application areas, like IoT [23], AI acceleration [13], and High-Performance Computing (HPC) [5]. At the same time, open-source RISC-V hardware models, design tools, and toolchains have been introduced, including PULP [22], OpenTitan [18], and Chipyard [8]. As a result, different RISC-V architectures covering single-core and multi-core platforms became available, such as PULPissimo, OpenPULP, and Hero, just to mention a few examples from the PULP platform [22].

Towards many-core architectures, the Grid of Processing Cells (GPC) platform has been proposed [9]. Its scalability has been demonstrated at the system level by the use of SystemC TLM-2.0 simulation [30].

This paper introduces a scalable and synthesizable GPC prototype based on the Rocket Chip system with RTL generation and simulation in the context of the Chipyard framework [8]. In contrast to the high-level GPC specification [9], we present here a scalable Chisel-based implementation for RTL synthesis of N modified RocketTiles using the Rocket Chip generator for Verilog generation [3]. To adapt RocketTiles to grid processing cells with local memories, we replaced the local data cache of each RocketTile with a scratchpad memory, linked them via the TileLink on-chip network, and modified the bootloader accordingly. We modified the Chipyard toolchain for multi-core software compilation and extended the RISC-V Frontend Server (FESVR) to load software to the scratchpad memories during the RTL simulation of the Verilog models. Our Rocket Chip based GPC prototype was functionally tested by the generation and simulation of a design with 4 RocketTiles, each executing a Hello-World program in parallel.

The remainder of this paper is structured as follows. After the related work in Section 2, a short introduction to the principles of the GPC platform is given in Section 3. Next, Section 4 presents the tool flow using the Chipyard framework, the Rocket Chip generator, and the Verilator. Section 5 introduces the implementation of an early GPC prototype based on Rocket Chip with modifications to the bootloader and software compilation. The required modifications to the FESVR and the simulation of a simple example are presented in Section 6. Finally, Section 7 closes with a summary and conclusion, followed by future work.

2 Related Work

The well-known *memory bottleneck* describes the traffic congestion of data and instruction streams through a single memory bus and has already been observed in the classic von-Neumann computer architecture [16, 12]. While today's computers are typically organized as symmetric multiprocessors (SMPs) [20] and feature multiple (or many) processing cores on a chip, SMPs still use a single shared interconnect to the main memory, which hinders true scalability.

To avoid the memory bottleneck, alternative many-core ar-

chitectures have been proposed and built in research environments. One example is the Raw Processor [26], which features a 4x4 tile architecture with multiple buses and separate memories. The architecture is scalable with increasing silicon density but it is limited to application-specific resource allocation, mapping, and data flow.

Another example is the many-core Tile Processor [31] by Tilera. Variants offer 8x8 cores (TILE64 [27] and TILEPro64 [28]) or 100 cores [7]. In all examples, each tile consists of a general-purpose processor with a cache and a router for inter-processor and I/O device communication.

Intel has implemented the Polaris research chip with a network-on-chip (NoC) architecture with 80 cores connected by a mesh network [29], which reportedly features sustained performance of 1.28 tera-FLOPS [21]. Another product is the Single-Chip Cloud Computer whose communication structures resemble a data center [11]. Here, each tile in the 4x6 mesh network contains two Pentium cores and a router. Intel has also fabricated many-core processors for use as co-processors in servers [25]. These Xeon Phi chips contain 60 physical cores with bi-directional ring interconnect, where each node is a 4-way hyper-threaded x86 processor. Unfortunately, the high parallelism suffers severely from the limited bandwidth to the external memory [15].

Multiple separate memories have also been investigated for many-core architectures. Examples include the KiloCore processor array [6], which features 1000 independent processors and 12 memory modules on a single chip, and Epiphany-V [17], which uses a cache-less memory model.

3 Grid of Processing Cells (GPC)

Traditional single-, multi-, and many-core computer architectures suffer from the memory bottleneck to a single shared main memory which can delay many-core processors for thousands of cycles due to bus contention despite sophisticated multi-level cache hierarchies [15]. As an alternative *scalable* computer organization, tiled network-on-chip architectures have been proposed with *separate* local memories. This work follows the idea of a Grid of Processing Cells (GPC) [9] where pairs of processors and memories are arranged on-chip in a two-dimensional array with only local interconnect. In essence, the typical use of an expensive multi-level cache hierarchy is here replaced by many on-chip memories, similar to the scratchpad memory (SPM) approach commonly used in embedded computer systems [19, 4].

The checkerboard variant of a GPC is shown in Figure 1. Processor cores C_{yx} and local memories M_{yx} are paired as cells and arranged in an alternating pattern so that every processor has access to four neighboring memories. Cores on the edges of the chip have access to off-chip memories or memory-mapped I/O devices.

Each cell in the grid consists of a fully equipped general-purpose processor, such as a RISC-V core, and its own local memory of substantial size and high speed (SRAM).

Conceptually, the checkerboard GPC communication can

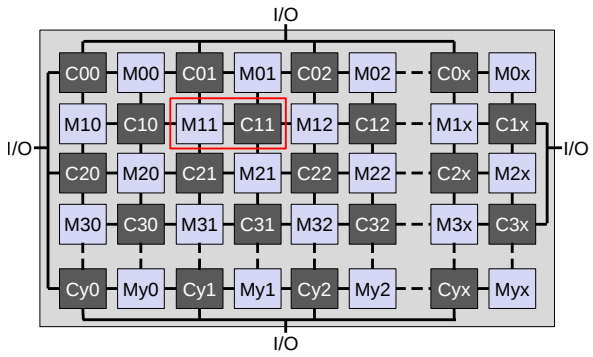

Figure 1 Checkerboard Grid of Processing Cells (GPC)

be established by a priority-based multiplexing interconnect within each cell, as shown in Figure 2. Here we use SystemC TLM-2.0 [14] initiator and target sockets arranged in multiplexer and de-multiplexer fashion to connect processors (initiators) with their neighboring memories (targets). To resolve concurrent access conflicts, priority-based arbitration can be implemented, giving each processor first priority access to its own local memory, and lower priority to access the memories in neighbor cells.

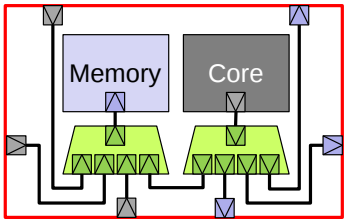

Figure 2 Checkerboard tile with SystemC TLM-2.0 interconnect

The checkerboard GPC has proven to be functional and scalable in system-level simulation [30]. Several embedded applications have been successfully mapped onto the GPC platform in 4×2 and 4×4 configurations, including a Canny edge detector and APNG encoder [10]. While high-level simulation with SystemC TLM-2.0 shows promising and scalable results, we now design and evaluate a detailed model in cycle-accurate Verilog.

4 Chipyard Based Tool Flow

This section introduces the Chipyard framework and the tools we used to design and simulate a RISC-V based System on Chip (SoC) prototype of the GPC platform.

4.1 Chipyard Framework

Chipyard is an open-source framework for the development, RTL simulation, FPGA prototyping, and VLSI implementation of RISC-V based SoCs, which has been introduced by UC Berkeley [1]. As such, Chipyard can be considered an all-in-one solution for RISC-V based SoC design, simulation, and synthesis [8].

Chipyard comes with the hardware construction language

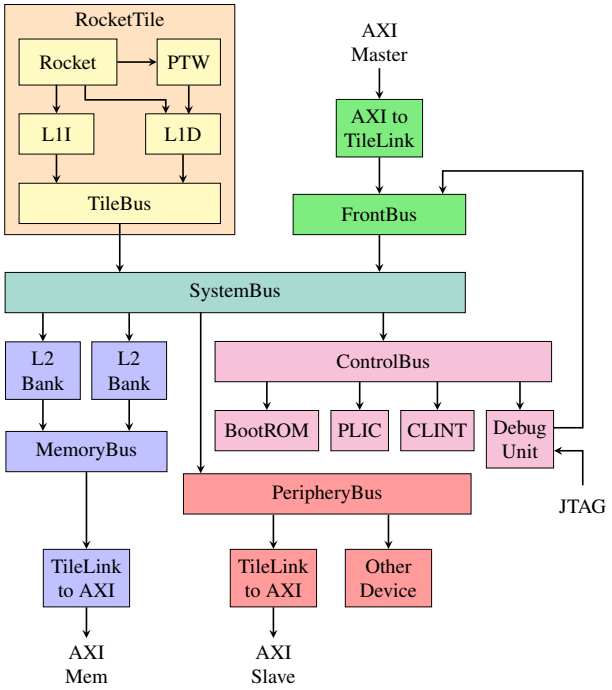

Figure 3 Typical Rocket Chip system based on [8]

Chisel [3] from which Verilog Hardware Description Language (HDL) code for all parts of an SoC, including processor cores, memory systems, and peripherals, can be generated. In our design, we applied a tool flow that generates Verilog RTL code using the Rocket Chip generator, which can then be simulated by Verilator. Finally, we adapted the Chipyard compiler toolchain to build a software application for the generated SoC.

The remainder of this section provides a brief introduction to the Rocket Chip generator followed by a description of the basic principles of the Verilator simulation as well as the software toolchain.

4.2 Rocket Chip Generator

The Rocket Chip generator is an SoC generator, written in the hardware configuration language Chisel [8, 2]. It generates scalable Rocket Chip systems and is the basis of the Chipyard framework. Figure 3 shows a typical Rocket Chip system with a single Rocket core. The Rocket core is an in-order RISC-V processor core with a five-stage pipeline. The core is embedded in the RocketTile, which connects the core to the first-level data and instruction caches, a page-table walker, and the TileBus connected to the SystemBus. The SystemBus is linked to the FrontBus, PeripheryBus, and ControlBus. Both the PeripheryBus and SystemBus allow memory-mapped access to attached peripherals. The bootloader is located in the BootROM attached to the ControlBus. On startup, all cores execute the bootloader instructions from the BootROM. Besides the BootROM, a platform-level (PLIC) and core-local (CLINT) interrupt controller and a Debug Unit are attached to the ControlBus. The Debug Unit allows debugging of the system via a JTAG interface and is attached to the FrontBus. Optional external peripheral devices can be attached to the PeripheryBus. External main memory can be attached to the MemoryBus, which can then be accessed via the second-level caches through the SystemBus. All buses in the Rocket Chip system are implemented using Berkeley's TileLink on-chip network protocol [24].

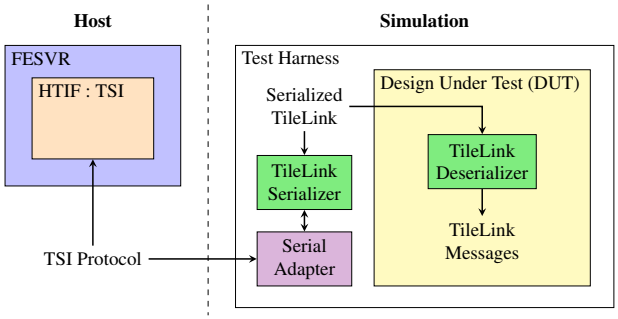

Figure 4 Communication with the DUT based on [8]

4.3 Verilator Simulator

Verilator is a cycle-accurate open-source Verilog simulator with waveform export in VCD format, which is used to conduct RTL simulations of the generated SoC models in Chipyard. The Verilator translates the generated Verilog code into optimized C++ code, which is then further compiled into a binary executable that conducts the actual simulation. When starting the simulation of the Rocket Chip system, the binary Executable and Linkable Format (ELF) file of the cross-compiled software gets loaded into the scratchpad memories from where the Rocket cores then fetch, decode, and execute the instructions.

For simulation, as shown in Figure 4, the Design under Test (DUT) and the host communicate via the Frontend Server (FESVR), a C++ library using the Tethered Serial Interface (TSI) protocol [8]. The FESVR is used to load the software binary into the DUT memory. It also intercepts and handles system calls from the simulated software, e.g., inputs/outputs from an executed program to the console of the simulator. In the test harness, a Serial Adapter module converts the TSI commands into TileLink requests. Those TileLink requests are serialized and forwarded to the DUT. In the DUT, the serialized TileLink requests are deserialized and routed via the FrontBus [8].

5 GPC Prototype with Rocket Cores

Our design of a GPC prototype in Chipyard required several modifications of the Rocket Chip system, which we present in this section.

5.1 Platform Architecture

As a starting point we used the standard Rocket Chip system shown in Figure 3. Our modification is mainly based on the reconfiguration and replication of RocketTile modules so they can become processing cells in the grid. Our current prototype design with N modified RocketTiles is shown in Figure 5.

To reduce the complexity, we restricted the Rocket cores

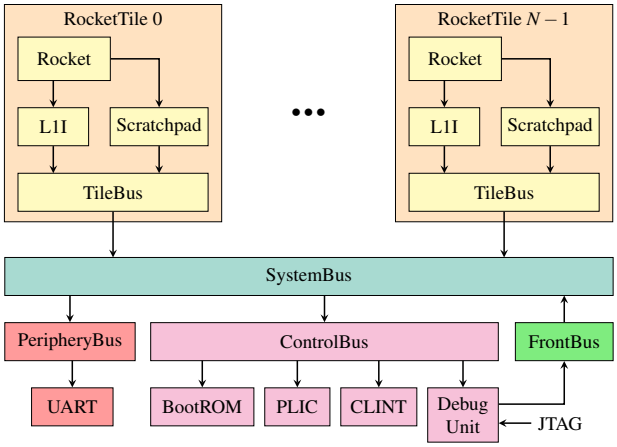

Figure 5 Modified Rocket Chip system with N RocketTiles

to the RV32IMA ISA subset and further modified the original Rocket Chip system. As the GPC platform uses a local memory in each processing cell, we replaced the first-level data cache with a scratchpad memory to become the local memory of the cell. Furthermore, we striped off other memory structures, which are not needed in the GPC platform, including the second-level cache, the external memory, the memory bus, and all AXI interfaces. Since we focus on bare-metal applications, we removed the page-table walker as well. With the modified RocketTiles, the configurable Rocket Chip generator allows the generation of an arbitrary number of processing cells. The Rocket Chip system example in Figure 5 has N processing cells, implemented by RocketTiles attached to the SystemBus, each with a local scratchpad memory. For serial output, the final architecture also includes a UART interface attached to the PeripheryBus.

As proposed by the GPC approach [9], we implement each local memory as scratchpad memory, which holds the complete text and data sections of the software to be executed by the core of the processing cell. To demonstrate the memory addressing scheme, an excerpt of the memory map of a configuration with $N = 4$ RocketTiles is shown in Table 1. Each scratchpad memory is configured to the size of 32 KiB to fit small demonstration programs. The address space of the scratchpad memory of the first RocketTile ranges from address 0x 8000 0000 to 0x 8000 8000 exclusively. All address spaces of the scratchpad memories of other RocketTiles are aligned without space between and ordered by the index n of the corresponding RocketTile. Thus, the address space of the scratchpad memory of the RocketTile with index $0 \leq n \leq N-1$ ranges

$$\begin{aligned} &\text{from} && (0x\,8000\,0000) + n \cdot (0x\,8000) \\ &\text{to} && (0x\,8000\,0000) + (n+1) \cdot (0x\,8000). \end{aligned}$$

The offset of $0x\,8000$ corresponds to the memory size of 32 KiB per scratchpad. By default, every access to memory apart from the own scratchpad of a core, e.g., when reading from the BootROM, is handled via TileLink. Therefore, a request is sent from the TileBus of the corresponding RocketTile to the SystemBus. The SystemBus sends the TileLink request to its destination based on the requested memory address. Thereby, in the current configuration, every Rocket core can access the scratchpad memories of all other cores. Depending on the specific GPC architecture, e.g., the checkerboard GPC, the access will be restricted to neighboring memories accordingly. However, in general, using the shared SystemBus to access neighboring scratchpads has to be avoided as it may introduce a bottleneck. Therefore, as proposed by the introduced GPC architecture, local interconnects for memory access between neighboring processing cells will be implemented in future designs. Due to the modular TileLink on-chip network of the Rocket Chip system, the routing of memory access requests can be customized easily. Thus, all GPC-related modifications to the memory infrastructure of a processing cell can be achieved by modifying the TileLink network. By implementing these additional interconnects, the checkerboard GPC and other variants [9] can be realized.

Base	Top	Device	Size
…	…	…	
0x 0000 4000	0x 0000 5000	BootAddrReg	4 KiB
0x 0001 0000	0x 0002 0000	BootROM	64 KiB
…	…	…	
0x 0200 0000	0x 0201 0000	CLINT	
0x 0c00 0000	0x 1000 0000	PLIC	
…	…	…	
0x 5400 0000	0x 5400 1000	UART	
0x 8000 0000	0x 8000 8000	Scratchpad 0	32 KiB
0x 8000 8000	0x 8001 0000	Scratchpad 1	32 KiB
0x 8001 0000	0x 8001 8000	Scratchpad 2	32 KiB
0x 8001 8000	0x 8002 0000	Scratchpad 3	32 KiB

Table 1 Memory map of the modified Rocket Chip system with $N = 4$ RocketTiles

5.2 Bootloader Modification

On startup, all Rocket cores boot from the same BootROM at address 0x1 0000. The default bootloader, which resides in the BootROM, would then jump to address 0x 8000 0000. As every core executes those bootloader instructions, all cores would jump to the scratchpad memory address of the first RocketTile by default. However, this behavior would only be desired for multi-threaded shared memory applications. For the GPC platform, each processing cell is supposed to execute instructions from its own local memory, which requires modifications to the default bootloader as shown in Listing 1. Lines 7 to 14 show the required modification of the bootloader. The code generates an offset depending on the hardware thread (hart) id of the individual core, which can be read from the `mhartid` Control and Status Register (CSR) of each core. The hart id is then loaded into the `t0` register (line 7) and the offset between the scratchpads into the `t1` register (line 8). The register `a0` already contains the address of the first

scratchpad. A loop decrements `t1` until it reaches zero. Each iteration of the loop adds the offset stored in `t1` to `a0`. After the last iteration, the `a0` register of each core contains the address of the respective scratchpad memory. That address is then written into the Machine Exception Program Counter (MPEC) CSR (line 15). When the `mret`-instruction is executed (line 20), the core jumps to the address stored in the MPEC CSR.

```
1   #define BOOTADDR_REG 0x4000
2   #define BOOTADDR_OFFSET 0x8000
3   ...
4   li a0, BOOTADDR_REG
5   lw a0, 0(a0)
6   ...
7   csrr t0, mhartid
8   li t1, BOOTADDR_OFFSET
9   beqz t0, 2f
10 1:
11  add a0, a0, t1
12  addi t0, t0, -1
13  bnez t0, 1b
14 2:
15  csrw mepc, a0
16  csrr a0, mhartid
17  la a1, _dtb
18  li a2, 0x80
19  csrc mstatus, a2
20  mret
```

Listing 1 Modification to the bootloader assembly code

5.3 Compilation and Execution of Bare-Metal Software

To build multi-core bare-metal RISC-V applications that can be executed by the generated SoC, we extended the Chipyard framework by a bare-metal software environment. The environment comes with a Makefile for the compilation of applications, the assembly code to set up the C runtime, a linker script, and libraries to communicate with the FESVR. As the Chipyard framework currently only includes a 64-bit RISC-V cross-compiler toolchain, we configured a custom compiler toolchain to cross-compile programs for our targeted RV32IMA 32-bit ISA.

To conduct a functional test of the GPC prototype and the software environment, we implemented a simple Hello-World C program, as shown in Listing 2. The C code of each core is enclosed by the `thread_entry` function. The `thread_entry` function is called with the individual hart id and the total number of harts as arguments after setting up the C run-time environment. The example uses a single statement, which prints the hart id of the core and the total number of cores. The compiled Hello-World program is loaded into the scratchpad memory of each RocketTile, as outlined in the following section.

```
1  void thread_entry(int cid, int nc)
2  {
3    printf("Hello from core %d of %d!\n",
4           cid, nc);
5    exit(0);
6  }
```

Listing 2 Multi-core Hello-World C program

6 GPC Prototype Simulation

By default, the simulator built with the Verilator simulator only supports loading one ELF file via the FESVR. The ELF file contains the binary executable and the base memory address where to load the data. That address is specified in the linker script, which is used by the linker. To load the program to the first scratchpad memory, the base address is set to 0x 8000 0000 by default.

To load an arbitrary number of ELF files to any given address, we extended the FESVR by adding an individual offset to the base address embedded in each ELF file. Hereby we support both, the loading of different ELF files and the loading of the same ELF file to different addresses.

The syntax of this `multiload` feature is shown in Listing 3. The `load-count` argument specifies how many times the ELF file located at `elf-path` is loaded into the memory. The first ELF file is loaded by adding the `base-offset` to the base address of the ELF file. Each following load increments the offset by the `offset-per-elf` argument. As a result, the n-th load of the ELF file is done by adding the offset

$$(\texttt{base-offset}) + n \cdot (\texttt{offset-per-elf})$$

to the base address.

```
1  ./simulator-build
   ↪ multiload=<load-count>,<base-offset>,
   ↪ <offset-per-elf>,<elf-path>
```

Listing 3 Syntax of the `multiload` command

As the introduced Hello-World program calls the `printf` function, the FESVR has to handle the corresponding system call. By default, the FESVR can only handle system calls of one core. For a multi-core simulation, we modified the FESVR to handle system calls of all cores separately.

For the RTL simulation of our Rocket Chip based GPC prototype and the modified FESVR, we configured a model with $N = 4$ RocketTiles and generated Verilog code, which was finally simulated by the Verilator. We cross-compiled

the Hello-World example and loaded it to the scratchpad memories of the 4 cores in the simulation.

Listing 4 shows the console output during the simulation. As specified by the `multiload` arguments (line 1), the ELF file `hello` is loaded $N = 4$ times by adding the offset

$$0\text{x}0 + n \cdot 0\text{x}\,8000$$

to the n-th load with $0 \leq n \leq 3$. The progress of loading the ELF file to the memories is shown in lines 4 to 7. After the last load, the Rocket cores start to execute the instructions of the bootloader. The modified bootloader causes each core to jump to its own scratchpad memory. When executing the Hello-World program, each core successfully prints its hart id and the total number of harts in the system (lines 8-11) as it is defined in the corresponding Hello-World program.

```
1  $ ./simulator-QuadGPCTinyRocketConfig
   ↪ multiload=4,0,0x8000,./hello
2  Listening on port 33501
3  [UART] UART0 is here (stdin/stdout).
4  Loading hello with offset     0x0 done
5  Loading hello with offset  0x8000 done
6  Loading hello with offset 0x10000 done
7  Loading hello with offset 0x18000 done
8  Hello from core 1 of 4!
9  Hello from core 2 of 4!
10 Hello from core 3 of 4!
11 Hello from core 0 of 4!
```

Listing 4 Output of the Verilator RTL simulation with $N = 4$ RocketTiles executing the Hello-World program

7 Summary and Conclusion

In this work, we have introduced an early Rocket Chip based prototype of the GPC platform using the Chipyard framework. This Chisel-based implementation allows generating models with an arbitrary number of processing cells for RTL simulation and synthesis. The processing cells are represented by modified RocketTiles with scratchpad memory connected via TileLink. For the Verilator simulation, we have extended the Chipyard toolchain by a bare-metal software environment to cross-compile C programs for a multi-core RV32IMA 32-bit RISC-V architecture. The modified RISC-V FESVR automatically loads the compiled software application to the individual scratchpad memories of the cores.

To validate our approach, we have configured a GPC prototype with 4 RISC-V processing cells and generated the RTL model with the Rocket Chip generator. Using the generated Verilog model, we have performed an RTL simulation with the Verilator simulator. A simple Hello-World application was compiled and successfully executed by all 4 RISC-V cores.

As a next step, we will focus on aligning the communication and memory architecture of our RTL GPC design with the GPC platform given in [9]. This mainly concerns direct memory access to neighboring GPC cells via TileLink to avoid the SystemBus bottleneck and different strategies for synchronization of communication between processing cells. In addition, we plan to introduce a virtual address space for each core to avoid handling the different physical address spaces of particular processing cells in software.

At last, we will investigate FPGA synthesis of our Rocket Chip based GPC to build a first hardware-based implementation that can execute GPC applications in real time. We also plan to test the scalability and throughput on an FPGA cluster with different applications.

Acknowledgements

The work described herein is partly funded by the German Bundesministerium für Bildung und Forschung (BMBF) through the Scale4Edge project (16ME0133). The authors would also like to thank Bastian Koppelmann for his valuable support.

References

[1] Alon Amid et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[2] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[3] Jonathan Bachrach et al. “Chisel: Constructing hardware in a Scala embedded language”. In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[4] Rajeshwari Banakar et al. “Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems”. In: *Proceedings of the International Symposium on Hardware-Software Codesign (CODES)*. ACM, 2002, pp. 73–78.

[5] Andrea Bartolini et al. “Monte Cimone: Paving the Road for the First Generation of RISC-V High-Performance Computers”. In: *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. 2022, pp. 1–6. DOI: 10.1109/SOCC56010.2022.9908096.

[6] Brent Bohnenstiehl et al. “KiloCore: A 32-nm 1000-Processor Computational Array”. In: *IEEE Journal of Solid-State Circuits* 52.4 (2017), pp. 891–902. DOI: 10.1109/JSSC.2016.2638459.

[7] Charlie Demerjian. *A look at the 100-core Tilera Gx*. https://www.semiaccurate.com/2009/10/29/look-100-core-tilera-gx/. [Online; accessed 30-May-2022]. Oct. 2009.

[8] *Chipyard Documentation*. Release 1.8.1. Oct. 2022. URL: https://chipyard.readthedocs.io/_/downloads/en/1.8.1/pdf/.

[9] Rainer Dömer. *A Grid of Processing Cells (GPC) with Local Memories*. Tech. rep. CECS-TR-22-01. UCI: Center for Embedded and Cyber-physical Systems, Apr. 2022.

[10] Vivek Govindasamy, Emad Arasteh, and Rainer Dömer. "Minimizing Memory Contention in an APNG Encoder using a Grid of Processing Cells". In: *Proceedings of the International Embedded Systems Symposium (IESS)*. Lippstadt, Germany: Springer, Nov. 2022.

[11] Jim Held. ""Single-chip Cloud Computer", an IA Tera-scale Research Processor". In: *Euro-Par 2010 Parallel Processing Workshops*. Ed. by Mario R. Guarracino et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 85–85. ISBN: 978-3-642-21878-1.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[13] Pouya Houshmand et al. "DIANA: An End-to-End Hybrid DIgital and ANAlog Neural Network SoC for the Edge". In: *IEEE Journal of Solid-State Circuits* 58.1 (2023), pp. 203–215. DOI: 10.1109/JSSC.2022.3214064.

[14] IEEE Computer Society. *IEEE Standard 1666-2011 for Standard SystemC Language Reference Manual*. IEEE, New York, USA, 2011.

[15] Guantao Liu et al. "Optimizing Thread-to-Core Mapping on Manycore Platforms with Distributed Tag Directories". In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. Tokyo, Japan, Jan. 2015.

[16] John von Neumann. *First Draft of a Report on the EDVAC*. Tech. rep. University of Pennsylvania, June 1945.

[17] Andreas Olofsson. "Epiphany-v: A 1024 processor 64-bit risc system-on-chip". In: *arXiv preprint arXiv:1610.01832* (2016).

[18] *OpenTitan*. lowRISC. Nov. 2022. URL: https://docs.opentitan.org.

[19] Preeti R. Panda, Nikil D. Dutt, and Alexandru Nicolau. "Efficient utilization of scratch-pad memory in embedded processor applications". In: *Proceedings of the European Design Automation Conference (Euro-DAC)*. 1997, pp. 7–11. DOI: 10.1109/EDTC.1997.582323.

[20] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012. ISBN: 978-0-12-374750-1. URL: http://www.elsevierdirect.com/product.jsp?isbn=9780123747501.

[21] Li-Shiuan Peh, Stephen W. Keckler, and Sriram Vangal. "On-Chip Networks for Multicore Systems". In: *Multicore Processors and Systems*. Ed. by Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee. Boston, MA: Springer US, 2009, pp. 35–71. ISBN: 978-1-4419-0263-4. DOI: 10.1007/978-1-4419-0263-4_2. URL: https://doi.org/10.1007/978-1-4419-0263-4_2.

[22] Pasquale Davide Schiavone et al. "Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX". In: *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2018, pp. 1–3. DOI: 10.1109/S3S.2018.8640145.

[23] Ronaldo Serrano et al. "A Low-Power Low-Area SoC based in RISC-V Processor for IoT Applications". In: *2021 18th International SoC Design Conference (ISOCC)*. 2021, pp. 375–376. DOI: 10.1109/ISOCC53507.2021.9613880.

[24] *SiFive TileLink Specification*. Version 1.8.1. SiFive Inc. Jan. 2020. URL: https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf.

[25] Avinash Sodani et al. "Knights Landing: Second-Generation Intel Xeon Phi Product". In: *IEEE Micro* 36.2 (2016), pp. 34–46. DOI: 10.1109/MM.2016.25.

[26] Michael Bedford Taylor et al. "The Raw Processor: A Composeable 32-Bit Fabric for Embedded and General Purpose Computing". In: 2001.

[27] Tilera. *Manycore without Boundaries: TILE64 Processor*. http://www.tilera.com/products/processors/TILE64. [Online; accessed 30-May-2022].

[28] Tilera. *Manycore without Boundaries: TILEPro64 Processor*. http://www.tilera.com/products/processors/TILEPRO64. [Online; accessed 30-May-2022].

[29] Sriram Vangal et al. "An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS". In: *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. 2007, pp. 98–589. DOI: 10.1109/ISSCC.2007.373606.

[30] Yutong Wang, Arya Daroui, and Rainer Dömer. "Demonstrating Scalability of the Checkerboard GPC with SystemC TLM-2.0". In: *Proceedings of the International Embedded Systems Symposium (IESS)*. Lippstadt, Germany: Springer, Nov. 2022.

[31] David Wentzlaff et al. "On-Chip Interconnection Architecture of the Tile Processor". In: *IEEE Micro* 27.5 (2007), pp. 15–31. DOI: 10.1109/MM.2007.4378780.