Implementation of Different Communication Structures for a Rocket Chip Based RISC-V Grid of Processing Cells

Lars Luchterhandt[1], Tom Nellius[1], Robert Beck[1], Rainer Dömer[2], Pascal Kneuper[1], Wolfgang Mueller[1], and Babak Sadiye[1]
[1]Paderborn University/Heinz Nixdorf Institute, Paderborn, Germany
Email: {larluc,beckr,tnellius}@mail.upb.de, {pkneuper,wmueller,babaks}@hni.upb.de
[2]University of California, Irvine, USA
Email: doemer@uci.edu

Abstract

There are currently multiple investigations of various RISC-V architectures for different applications from single-core based edge processors to powerful multi-core systems for High-Performance Computing (HPC). We have recently introduced a many-core GPC (Grid of Processing Cell) architecture based on configurable modified RISC-V RocketTiles with local memories and systembus communication, which was demonstrated and evaluated using RTL simulation by simple distributed software programs [17]. In this work, we have refined and extended our development by two communication schemes for direct intercell processor communication between neighboring cells: via (i) shared memory and via (ii) RISC-V Control and Status Registers (CSRs). Both alternatives are implemented on an AMD VCU108 FPGA and as chip layouts based on SkyWater 130 nm CMOS technology. The performance is evaluated by a distributed systolic matrix multiplication algorithm with different grid sizes.

1 Introduction

The open standard RISC-V instruction set architecture (ISA) has received worldwide acceptance as a viable commercial alternative for microprocessor-based systems. Many implementations and variants are currently under investigation covering multiple application areas, like IoT [23], AI acceleration [14], and High-Performance Computing (HPC) [5]. Meanwhile, multiple open-source RISC-V hardware models, design tools, and toolchains have been introduced for RISC-V development, like PULP [22], OpenTitan [19], and Chipyard [8]. Also, different RISC-V single-core and multi-core architectures with various SoCs have been introduced since then, like PULPino, PULPissimo, OpenPULP, and Hero [22], just to mention a few examples from the PULP platform.

Towards many-core architectures, the Grid of Processing Cells (GPC) platform has been proposed [9]. Its scalability with different applications has been demonstrated at the system level [28] and instruction-set level [12]. In prior work [17], we have introduced a first scalable Chisel-based [3] GPC prototype in the context of the Chipyard framework [8]. Our Chisel implementation modified the given RocketTile for grid processing cell configurations to generate arbitrary sizes of GPC mesh and torus structures. We introduced an architecture that replaced the local data cache of the RocketTile with a local scratchpad memory and linked the tiles to the TileLink network of the systembus, for IO, BootROM, and intercell communication [17].

In this paper, we refine the purely systembus-based communication structure of the RocketTiles and introduce additional TileLink interconnects to directly access the Data Tightly Integrated Memory (DTIM) of neighboring cells and to avoid non-scalable systembus communication. For intercell communication, we present two different handshake alternatives. Our software-based solution synchronizes the communication via shared memory, while the hardware-based alternative uses custom RISC-V Control and Status Registers (CSR). We compare the resource and area utilization of both alternatives with different grid sizes for FPGA and chip layouts. Our FPGA synthesis utilizes VIVADO with an AMD VCU108 FPGA board. Our new chip layouts are implemented using Cadence Genus/Innovus based on SkyWater 130 nm CMOS technology. Both handshake solutions are evaluated by a software program of a systolic matrix multiplication. We compare the bare metal execution of distributed software binaries on different GPC configurations by Verilator simulations.

Building upon our previous research presented in [17], this paper establishes a foundation by revisiting some key concepts in the first sections. Following a brief discussion of related work in Section 2, we provide a short introduction to the GPC platform in Section 3. Section 4 presents an overview of the Chipyard framework, Rocket Chip architecture, TileLink communication, and the Tightly Integrated Memory (TIM) interface. Our processing cell architecture with different interconnects is introduced in Section 5, while the FPGA and chip implementations are described in Section 6. The evaluation of a distributed software program on various grid sizes is then presented in Section 7, before closing with a summary and conclusion in Section 8.

2 Related Work

With the advent of multi-core and multi-processor architectures, a multitude of parallel processing platforms have been proposed and implemented. Following Flynn's taxonomy [10], different systems are distinguished by their instruction and data streams. In contrast to traditional uni-processor systems with single instruction single data (SISD) streams, modern architectures process multiple instruction and/or multiple data streams in parallel, termed as SIMD, MISD, or MIMD, respectively. Today, SIMD processor architectures are dominant and well supported as we can find them also directly supported by RISC-V P standard extensions. In this work, we are interested in MIMD machines, which promise the highest performance and flexibility. MIMD architectures come with either shared or distributed memories, and often with configurable data access structures.

The Grid of Processing Cells (GPC) architecture [9] that we implement in this work classifies as a MIMD machine with distributed memory (also known as *massively parallel processors* in the past). Recent examples of such many-core architectures include the scalable Raw Processor [26], which features a 4x4 tile architecture with multiple buses and separate memories, and the Tile Processor [31] which features up to 100 cores [7].

Products from Intel include the Polaris research chip with a network-on-chip (NoC) architecture of 80 cores connected by a mesh network [27], the Single-Chip Cloud Computer whose communication structures resemble a data center [13], and co-processors for use in servers [25], such as the Xeon Phi series with 60 physical cores and a bi-directional ring interconnect, where each node is a 4-way hyper-threaded x86 processor. Unfortunately, the high parallelism in these systems often suffers from limited bandwidth to a shared external memory [16].

Multiple separate memories have also been investigated for many-core architectures. Examples include the Kilo-Core processor array [6], which features 1000 independent processors and 12 memory modules on a single chip, and Epiphany-V [18], 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 [16]. As an alternative *scalable* computer organization, tiled network-on-chip architectures have been proposed with *separate* local memories.

This work implements 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. The typical use of an expensive multi-level cache hierarchy is here replaced by many on-chip memories, similar to *scratchpad* memories found in embedded computer systems [20, 4].

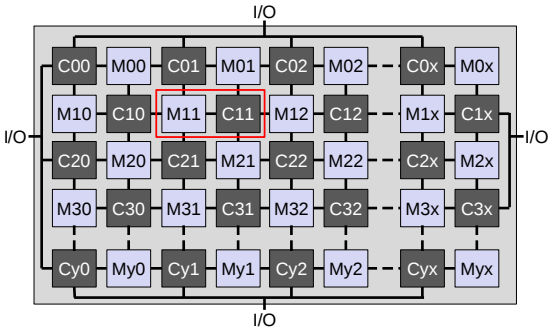

Figure 1 Checkerboard Grid of Processing Cells (GPC) [17]

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 be established by a priority-based multiplexing interconnect within each cell, as shown in Figure 2. The illustration uses SystemC TLM-2.0 [15] 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 is implemented, giving each processor first priority access to its own local memory, and lower priority to access the memories in neighbor cells.

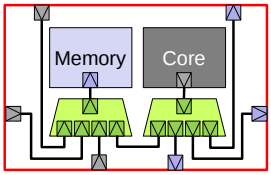

Figure 2 Checkerboard tile with SystemC TLM-2.0 interconnect [17]

The checkerboard GPC has proven to be functional and scalable in system-level [28] and instruction-level [12] simulation. Several embedded applications have been successfully mapped onto the GPC platform various configurations, including a Canny edge detector, APNG encoder [11], and a GoogleNet image classification Convolutional Neural Network (CNN) [21]. 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. We then synthesize this model to an FPGA prototype and finally as an actual chip layout.

4 Chipyard and Rocket Chip

Chipyard, developed by UC Berkeley [2] [8], is an open-source framework designed for the development, RTL simulation, FPGA prototyping, and VLSI implementation of RISC-V based SoCs. It utilizes the hardware construction language Chisel [3] to generate Verilog Hardware Description Language (HDL) code for various components within an SoC, such as processor cores, memory systems, and peripherals.

The Verilator simulator can simulate the generated Verilog code on a host computer which includes a compiler toolchain for the RISC-V ISA. Alternatively, the RTL can also be simulated on an FPGA for high-speed execution (FireSim). After simulation, the Verilog RTL is synthesized for FPGAs or it can be transferred to a chip layout by commercial tools from Synopsys and Cadence. In our design, we applied the following flows: Verilog generation, host simulation, FPGA simulation, FPGA synthesis, and layout generation for Sky130 130 nm CMOS technology.

4.1 Rocket Core

The Rocket Chip generator comes with a RocketTile and several SoC components which are written in the hardware construction language Chisel [3]. It can generate five-stage, in-order RISC-V processor cores called Rocket Core, which are embedded in a RocketTile. In the tile, the core is connected to the first-level data and instruction caches, a page-table walker, and the TileBus. The TileBus is linked to the SystemBus and connects via other buses to the other components like UART, BootROM, and Debug Unit.

4.2 TileLink Communication

Rocket Chip buses are typically implemented by TileLink hierarchical interconnect networks, which make use of directed acyclic graphs to guarantee deadlock-free communication [24]. Each point-to-point connection within the network is established by a link. Each link connects a client node of one module to a manager node of another module with only one path between the two nodes. Each link has two or more parallel channels depending on the conformance level (see Figure 3).

TileLink comes with three conformance levels [24]:

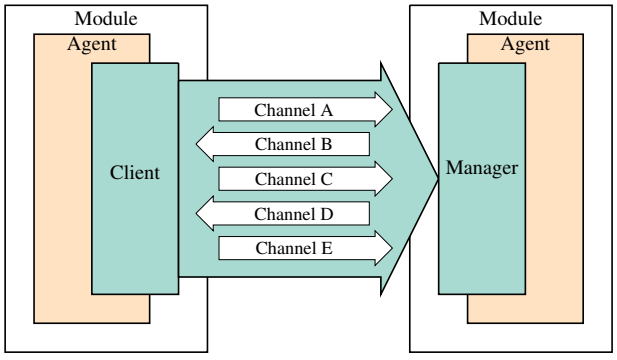

Figure 3 Links between Client and Manager module [24]

TileLink Uncached Lightweight (TL-UL), for simple, single-word Get and Put operations. TileLink Uncached Heavyweight (TL-UH), which adds features like atomic operations, burst access, and hints. TL-UL and TL-UH only require two of the five channels, channel A for requests from client to manager and channel D for the response. TileLink Cached (TL-C) introduces coherent caching, which requires the other three channels B, C, and E as shown in Figure 3.

Each module can contain multiple manager and client nodes for different purposes. Channels are unidirectional and carry TileLink messages while the communication itself is implemented via TileLink operations. Operations consist of several messages that propagate through the network in order to fulfill the operation. A TileLink operation is always issued by a client that sends messages to the manager node. The manager processes the received messages either by redirecting them to a client node of the same module or by answering them, if possible.

Rocket Chip implements buses as TileLink crossbars, like the SystemBus and the ControlBus. Crossbars have manager nodes that client nodes of higher hierarchies can link to. Internally, a crossbar is responsible for routing requests by forwarding an operation to the manager by a client node. Additionally, inside each Rocket tile, there is a collection of crossbars collectively named TileBus with a master crossbar and a slave crossbar.

4.3 Tightly Integrated Memory (TIM)

Rocket Chip systems implement the on-chip memory inside the data and instruction caches and scratchpad memories as Tightly Integrated Memories (TIM). They are called Data Tightly Integrated Memory (DTIM) or Instruction Tightly Integrated Memory (ITIM) due to their application for data or instruction storage, respectively.

Rocket Chip uses DTIM and ITIM adapters to translate the low latency protocol to TileLink messages, e.g., to connect scratchpad memory to the SystemBus. Multiple DTIM ports connected to a single DTIM are managed with a priority-based arbitration policy.

If an adapter with high priority is accessing data, the operations from lower-priority adapters are stalled until all higher-priority operations are handled, which poses the risk of starvation. However, it is possible to use different predefined arbitration policies, like round robin, which may process single requests from each requester in each round of requests.

5 Rocket Chip Based Grid of Processing Cells

Our GPC design required several modifications of the Rocket Chip architecture which is based on RocketTiles as single scalable processing cells with the final goal of combining them to arbitrary grid structures. We first give an overview of the configurability of our GPC cells. Thereafter, we present details of the individual processing cells and their interconnects.

In a first step, we modified a standard RocketTile for local memory cell processing. In the second step, we refined their local and global TileLink/DTIM intercell communication structure to directly connect to neighboring cells and introduced a virtual memory scheme for intercell communication.

5.1 GPC Configurations

Our Rocket Chip based implementation supports the configuration of arbitrary n-dimensional structures of RISC-V processing cells. The upper limit of the structure is currently only given by 32 communicating neighboring cells which is due to the size of an internal register. However, this is not a limitation of the basic concepts of the here presented approach. The GPC configuration needs to be defined in the Chisel implementation and thus be determined before the generation and synthesis of the hardware.

Though many different arbitrary cell structures are possible, we have limited our investigations to 2- and 3-dimensional grid structures so far. Figure 4 shows the example of a two-dimensional generic grid of processing cells connected to a torus of size $w \times h$. In this configuration, each Rocket core has access to the five address spaces shown in Table 1. The tiles located at the edges of the grid can be configured with additional torus connection to the tiles on the opposite side. Based on the hardware thread id (hartid), the relative position of a tile in the grid can be determined by the software.

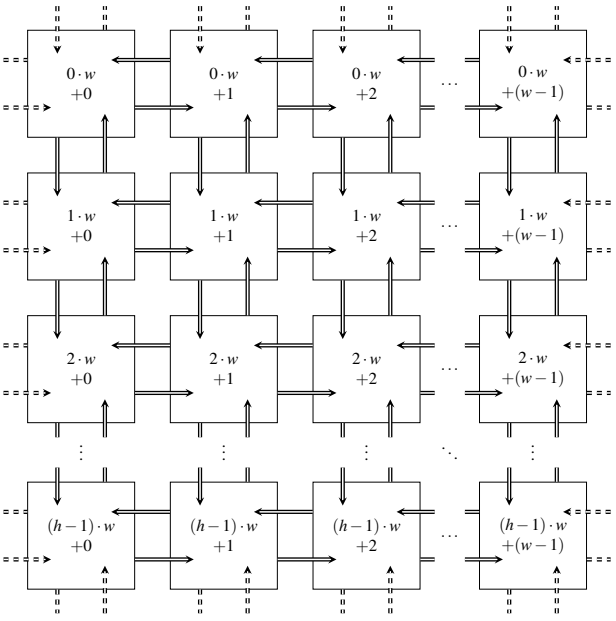

Figure 4 A GPC architecture with width w, height h, and optional torus connections

5.2 RocketTile Modifications

We restricted the Rocket cores to the RV32IMA ISA subset to reduce the cores complexity. To represent the local memory of each cell, we replaced the first-level data cache with a scratchpad memory. As other memory components are not needed, we removed the second-level cache, the external memory, the memory bus, the page table walker, and all AXI interfaces.

The size and starting address for the scratchpads are configurable. In our work, the scratchpad size is configured to 32 KiB which is sufficient for small demonstration programs. All RocketTile scratchpad memories are mapped without any gaps starting from 0x 8000 0000 and sorted by their hartid. Thus, the local memory of a RocketTile n is mapped

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

In a standard Rocket Chip multi-core SoC, memory accesses of each tile is handled via TileLink over the shared SystemBus. This includes access to the BootROM, memory-mapped devices, the own local memory, and memories of other Tiles. For many-core SoCs with local memory, this leads to a huge congestion of the SystemBus. To overcome the SystemBus bottleneck when accessing a local RocketTile scratchpad, we introduced an additional DTIM adapter for each RocketTile that connects the instruction cache directly to the scratchpad within the tile. Thus, the instruction cache only requires SystemBus access if instructions are not located in the scratchpad of the tile, e.g., as part of the BootROM attached to the SystemBus.

Additionally, the GPC platform requires direct access from each tile to the memory of neighboring tiles. However, as the neighboring relation should be highly configurable at synthesis, this requires special individual local configuration to neighboring cells. Therefore, each connection from a source tile to access the memory of a neighboring destination tile is implemented by one additional DTIM adapter in the destination tile, connected to the scratchpad memory of the destination tile. The source tiles can be connected to those DTIM adapters via TileLink and can thus access the neighboring scratchpad memories (see Figure 5). As access to a single scratchpad may be requested from multiple sources, there has to be an arbiter to manage those accesses. Currently, the arbiter is configured to use a priority-based arbitration policy. Here, the local core has the highest priority to the local scratchpad, followed by the DTIM adapter of the local instruction cache and the other neighbors.

5.3 Virtual Memory

To abstract from the system-wide physical addresses of the scratchpad of a tile, we introduced a virtual local addressing scheme for the core of each tile, which maps the address space of the neighboring scratchpad memories into a virtual address space of each tile. Thereby, from a core's perspective, its own local scratchpad memory is located at a fixed virtual address. The scratchpad memories of the core's configured neighbors are mapped consecutively in its virtual address space as shown in Table 1. As all cores apply this virtual addressing scheme, all cores refer to their own scratchpad with the same virtual address range. The scratchpad memories of neighboring tiles can be accessed by the relative position to their own tile without knowing

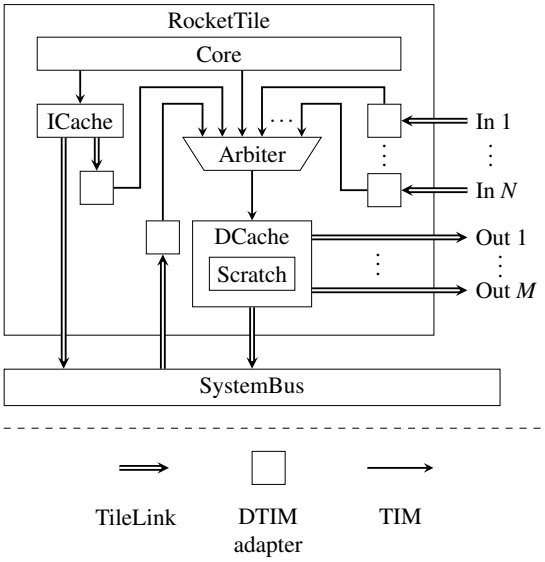

Figure 5 DTIM adapters in a tile with N incoming and M outgoing connections to neighbors

about their system-wide physical address space. This allows a programmer to write programs without knowledge about the physical memory address of each core in the grid.

Base	Top	Device	Size
0x 8000 0000	0x 8000 7FFF	Local Tile	32 KiB
0x 8000 8000	0x 8000 FFFF	Neighbor 0	32 KiB
0x 8001 0000	0x 8001 7FFF	Neighbor 1	32 KiB
0x 8001 8000	0x 8001 FFFF	Neighbor 2	32 KiB
0x 8002 0000	0x 8002 7FFF	Neighbor 3	32 KiB

Table 1 Local virtual memory map of each core in a GPC System with $N = 4$ neighbors

In standard Rocket Chip designs, a Translation Lookaside Buffer (TLB) is connected to the core in each tile. This TLB has the purpose of caching recent translations from virtual to physical memory addresses. In our design, there is currently no need for using a fully featured TLB. However, the TLB is still responsible for blocking access to invalid or protected addresses by granting or denying address requests from the core. To allow access to the introduced virtual addresses, the logic functions responsible for checking the validity of addresses inside of the TLB had to be extended. To access a core's own scratchpad memory via a virtual address, the hit function as part of the data cache had to be modified to indicate a hit to a core's scratchpad for addresses in the virtual address range instead of the physical address range. Since the SystemBus still needs to access each scratchpad memory by its physical address, the DTIM adapter that attaches the SystemBus to a scratchpad is used to map from the physical address to the virtual address which causes the hit function to indicate a hit to the scratchpad.

To access the scratchpad memories of neighboring tiles, additional DTIM adapters are used to establish TileLink connections to the neighboring scratchpad memories. Those DTIM adapters are extended by logic functions to map from the desired virtual address to the destination address of the neighboring scratchpad. As each individual tile has to offer scratchpad access to different neighboring tiles, there has to be an arbitration policy to manage multiple accesses. The current arbitration policy for local scratchpad access is priority-based. However, the individual scheme may depend on the GPC configuration and is subject to future investigations, e.g., to extend it to round-robin and others.

Each RocketTile has at least three DTIM ports connected to its own scratchpad memory. One for the core itself, one for the SystemBus used for loading and debugging, and one for fetching instructions by the instruction cache. While the core has a port for direct scratchpad memory access, the SystemBus and instruction cache are connected using DTIM adapters. Therefore, there are $2+N$ DTIM adapters attached to each scratchpad memory in our GPC approach as shown in Figure 5, where N is the number of neighbors requiring access to the scratchpad.

5.4 Processing Cell Interconnects

A basic example of a simple 1×2 grid structure with two interconnected tiles is shown in Figure 6. Both tiles have an additional DTIM adapter for the neighboring tile to access the local scratchpad memory. The breakthroughs from RocketTile 0 to RocketTile 1 and vice versa are implemented as a TileLink connection from one RocketTile via

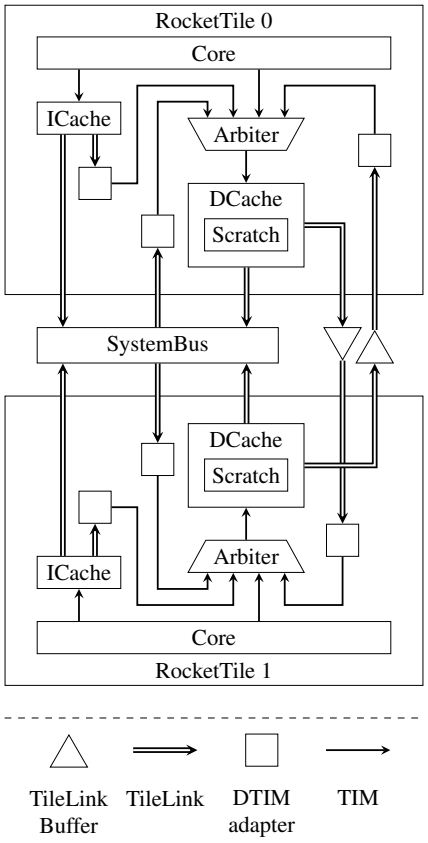

Figure 6 Two RocketTiles with mutual scratchpad access

a TileLink buffer to the additional DTIM adapter of the RocketTile neighbor. These DTIM adapters scale with the number of neighbors they need to connect to. The TileLink buffers are required to avoid combinational loops in the design and cause a one-cycle delay to the TileLink transaction. Thereby, a Rocket core can access the scratchpad memory of a neighbor by adding an additional DTIM adapter along with a TileLink connection.
The request of a core to one of its neighboring scratchpads via the virtual memory address range is first sent from the core to its data cache. As this virtual destination address does not cause the own scratchpad to indicate a hit, the request is forwarded from the data cache via TileLink to the DTIM adapter in the neighboring tile, for the virtual address. Inside this DTIM adapter, the requested virtual address is mapped to the address range served by the destination scratchpad memory. This request is then served by the destination scratchpad memory and the corresponding response is forwarded back to the requesting core.

5.5 Handshake-Based Communication

To communicate via neighboring scratchpad memories, we established a handshake-based communication scheme[1]. Thereby, the access is forced to happen synchronously with both the sender and receiver being actively involved.

5.5.1 General Principles

This handshake is performed by *ready* and *valid* indicators, similar to ARM's AXI interface [1]. The handshake procedure is given by the sequence diagram in Figure 7.
For the transfer to occur, both cores have to actively wait for a handshake from the respective communication partner. When a sender (Tile A) initializes the transfer to a receiver (Tile B), the sender has to wait for the receiver to be ready. Once Tile B confirms with a ready signal, Tile A can write the data to the shared memory in its own scratchpad, after which the valid signal is raised. The valid signal confirms to Tile B that data are available for reading. After reading, Tile B clears the ready signal and waits until Tile A resets the valid signal, which completes the procedure for Tile B. Tile A completes the procedure when receiving the reset of the ready signal, which is answered by resetting the valid signal.

5.5.2 Software Solution

For a software-based handshake synchronization for neighboring cell communication, we defined an additional `share` section in the linker script of our bare-metal environment which can be accessed by a fixed offset to the memory base address. Data transfer is established by a `share` struct with `valid`, `ready`, and `data` components.To use this data transfer in applications, we developed a small software library. The library provides send and receive functions along the synchronization scheme given at the beginning of this section.

[1]Here communication is tightly synchronized, in contrast to a looser push-pop buffer protocol proposed in [21].

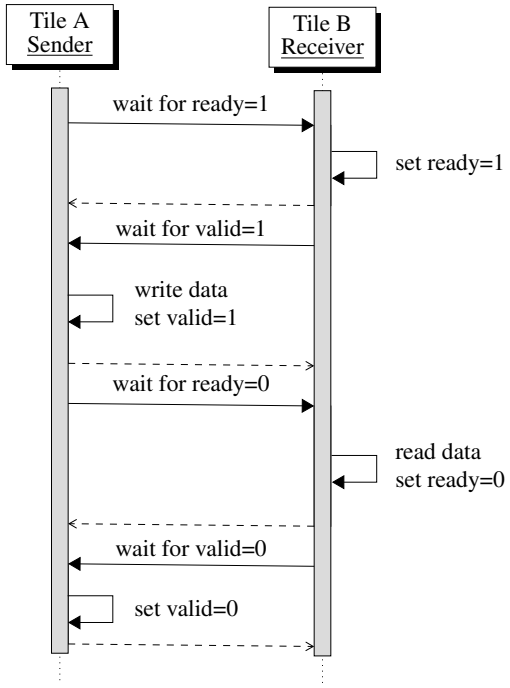

Figure 7 Handshake between sender and receiver

5.5.3 Hardware Solution

In the software approach, polling the `ready` and `valid` flags of other tiles in a loop results in extensive TileLink communication, long delays, and stall cycles. This can be avoided by the use of 1-bit communication lines that directly map to a CSR (Control and Status Register) with explicit signal lines to perform the handshake. When enabled, two 1-bit connections with opposite directions are generated along with each outgoing TileLink edge as shown in Figure 8. Here, each tile has four additional CSRs. The `selfready` and `selfvalid` CSRs can be read and written while the `otherready` and `othervalid` CSRs can only be read. Only one bit in each of these four CSRs is wired to its

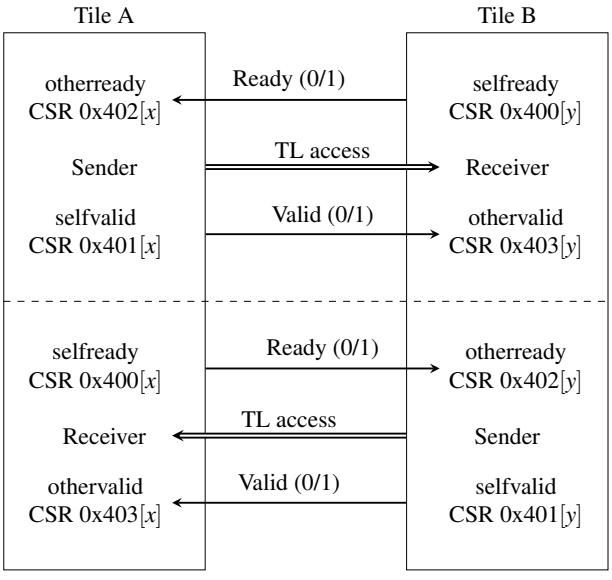

Figure 8 Additional ready and valid lines for handshakes

counterpart in the neighboring tile. The position of those bits is chosen according to the arrangement of neighboring cells and the virtual memory map per cell. So in Figure 8, Tile A uses the bit with index x in all four registers to access the signals wired to Tile B, where x corresponds to the relative position of Tile B to Tile A. In return, Tile B refers to the bits wired to Tile A with index y. Because of this mapping, the current limit for the number of neighbors for each tile is 32 when the Control and Status Register (CSR) connections are used. However, this could be extended by using additional CSRs. Next to each of the four CSRs, the addresses of the CSRs are indicated, starting from $0x400$.
The RISC-V ISA provides instructions to atomically read and set individual bits of CSR registers [29]. As these CSRs can be read and set in one instruction cycle while the signal lines do not use buffers in between, the changes occur at the respective neighbors immediately. Thereby, the repeated sampling of the ready and valid signals can be done instantly and without TileLink and its overhead while sending and receiving the data itself is still done via TileLink. In our studies, this improvement reduces the best-case communication time by roughly 12 cycles. However, this is just a first solution and requires further investigations in this context as the data transfer rate can be additionally improved by the use of scalable application-specific communication buffers between tiles.

6 Hardware Implementations

We applied different Chipyard flows to verify and generate hardware. For hardware generation, we have configured grids with 2×2 up to 8×8 cells to compare the impact of the different grid sizes and design configurations.
From Chisel we first translated RTL Verilog code via FIRRTL for Verilator simulation and for FPGA accerelated FireSim simulation. Thereafter, we applied the Chipyard FPGA prototyping flow which synthesizes Verilog code for FPGA boards using AMD Vivado and we applied the VLSI design flow using Cadence Genus and Innovus.

6.1 FPGA Implementation

The Chipyard FPGA flow with AMD Vivado required some extension to scale with larger GPC architectures. We thus extended the Chipyard FPGA prototyping support for the *AMD Virtex VCU108* and the *AMD Virtex VC707* FPGA board. We further implemented JTAG shells to upload and debug software on the FPGA boards which are connected to accessible GPIO port pins of the individual board. The software was finally loaded to the scratchpads via a JTAG debugger.
We synthesized different *N*x*N* grid configurations with CSR handshakes enabled and torus connections disabled to compare their Lookup-Table (LUT) utilizations. The results which synthesized at 100 MHz on a AMD VCU108 are given in Figure 9.
Each bar represents a different grid configuration. The SoC overhead of each design is colored green while the cores sizes are given in blue. We can see that the size in LUTs almost linearly increases with the number of cores.

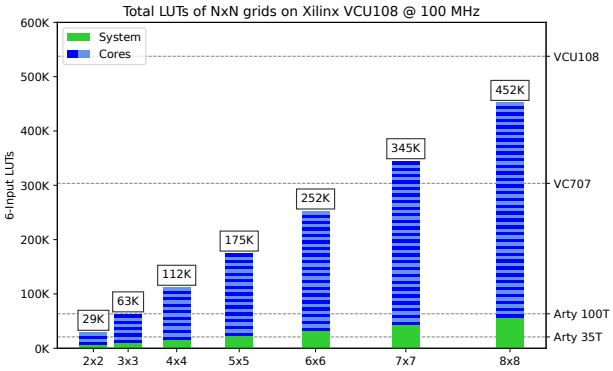

Figure 9 Different NxN grid configurations with #LUTs on AMD VCU108 at 100 MHz

The plot in Figure 9 indicates the LUT limitation of the used FPGA board. Note here that the 8×8 grid configuration takes 84% of the available LUTs on the biggest FPGA board (VCU108) in our evaluations.

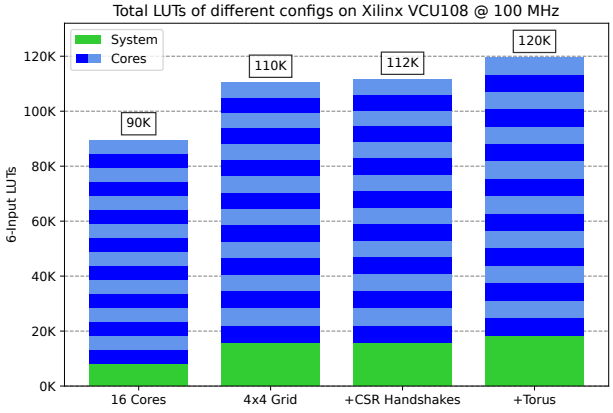

Figure 10 4×4 grids with #LUTs of different communication structures on AMD VCU108 at 100 MHz

Figure 10 shows the impact of different communication structures by the example of a 4×4 processing grid with four different configurations: (i) 16 single cores without grid overhead, (ii) 4×4 grid with modified RocketTiles and modified TileLink/TIM communication, (iii) 4×4 grid with modifications and additional CSRs for handshake support, and (iv) 4×4 grid with modifications and CSRs and additional torus connections. While the overhead per core only slightly increases when using a 4×4 GPC instead of 16 single cores, the SoC overhead almost doubles. The results also show that the introduction of handshake CSRs did not have any significant impact. The additional torus interconnects just had little impact, as each core at the edge requires one additional DTIM adapter and two additional TileLink connections.

6.2 Chip Implementation

The Chipyard VLSI flow was applied with Cadence Genus for logic synthesis and Innovus for Place & Route using SkyWater 130 nm technology. We generated GPC configurations from 2×2 upto 8×8 grids with CSR handshakes

and synthesized them using Cadence Genus at 50 MHz. Along with our observations from the FPGA studies, we can see in Figure 11, that the chip area scales linearly where the system overhead is almost negligible.

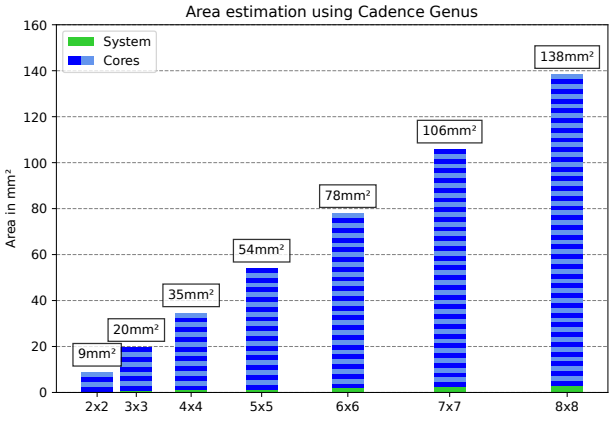

Figure 11 Total area after synthesis using Cadence Genus

Figure 12 shows the chip area utilization of a single RockeTile. It shows that about 85% of the chip area goes into memory which consists of 2×16 KiB DCache, a 32 bit ICache and a tag memory. The RISC-V Rocket core itself only uses 7.3% and the bus infrastructure and global systems take up 4.4% of the chip area with just a small portion of 2.5% for the interconnects and associated logic.

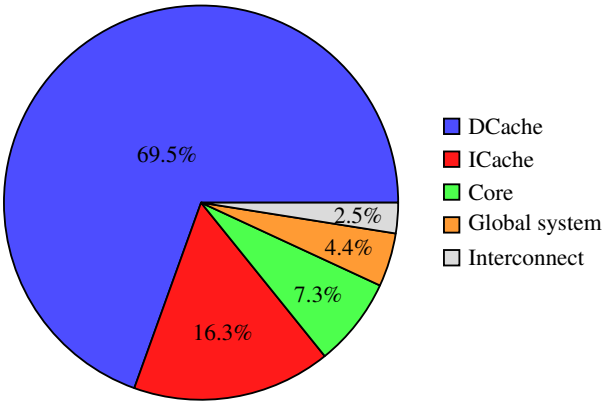

Figure 12 Average relative area per tile in a grid

Along our FPGA studies with different 4×4 grid configurations in Figure 10, we synthesized 16 single cores and a 4×4 grid with CSR handshakes and torus connections. The result is shown in Figure 13. When comparing the areas of the four configurations, it can be seen that in contrast to the FPGA synthesis the difference is almost negligible. This is caused by the fact, that almost 86% of the core area goes into memory. The communication structures with their interconnects have no significant impact on the total chip area.

Figure 14 shows the Cadence Innovus layout of the 4×4 grid layout with 16 cores and torus interconnects with a final area of 36.88 mm^2. The layout was implemented with our script for SRAM placement and rotation.

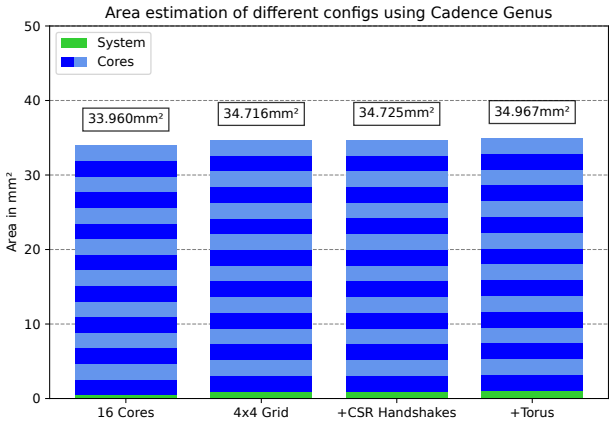

Figure 13 Total area of 4×4 grids with different communication structures

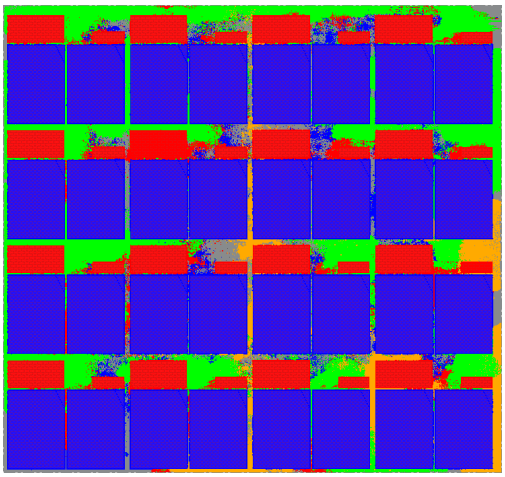

Figure 14 4×4 GPC chip layout using Cadence Innovus

7 Evaluation

We compiled different C programs to check the functional correctness and evaluated the performance by RTL Verilator simulations. To load the compiled software to the local scratchpad memory of the individual grid cells, we extended the RISC-V Frontend Server (FESVR) [17]. First functional tests implemented simple Hello-World programs with output via the systembus and simple exchanges of data tokens to test the communication of our mesh and torus structures. In addition to those simple software tests, we finally implemented a systolic matrix multiplication, which takes full advantage of scalable processing structure of our RISC-V grid as shown in Figure 15. For the systolic matrix multiplication each cell in the grid computes a partial multiplication result from the data received from its upstream neighbors, stores the result locally, and passes it to the downstream neighbor. Our experiments investigated different $N \times N$ grid configurations with $N = 1, \ldots, 8$ for which we have generated different hardware grids for RTL simulation and configured our software programs correspondingly.

The performance of the experiments was measured in terms of cycles by reading the `mcycle` CSR, which counts

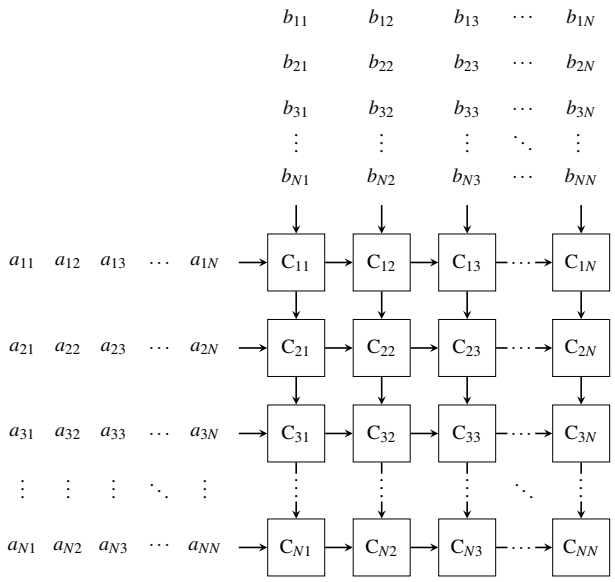

Figure 15 Systolic multiplication of two $N \times N$ matrices A and B on a $N \times N$ grid

the elapsed clock cycles [30]. As the `mcycle` CSR is an RTL register it can be used as a common base in simulation and in the synthesized FPGA hardware for performance measures. The final runtime on the FPGA is given by scaling it with a 100 MHz frequency for the hardware implementations in our previous section.

The systolic matrix multiplication algorithm is basically a batch processing application of several matrices where the values are automatically streamed through the processing cells. Our experiments took two matrices for the multiplication which we supplied through the systembus. Due to the structure of the algorithm, the matrices have the same dimensions as the respective grid.

Size	SW Handshake	HW Handshake	Speedup
2×2	1950	1734	1.12
3×3	4072	3498	1.16
4×4	6402	5846	1.10
5×5	9201	8331	1.10
6×6	11608	10563	1.10
7×7	14434	13488	1.07
8×8	17555	15452	1.14

Table 2 CPU cycles of the systolic matrix multiplication for different grid sizes

The evaluation results in terms of clock cycles are given in Table 2, which compares the speedup of the software handshake (SW) and the hardware handshake implementation via CSRs. In general, the hardware-based handshake implementation results in a moderate speedup between 7% and 16% with an average of 11%. Here, the table does not indicate a distinct dependence between the grid size and the respective speedup. As the resulting gain in speedup comes at a negligible hardware overhead, the hardware-based handshake offers a significant improvement in performance. In addition, any software-based handshake implementation can still be utilized without significant trade-offs. However, we still need to further investigate more efficient hardware communication schemes on larger data sets.

8 Summary and Conclusion

We have introduced a grid of processing cell architecture based on RISC-V Rocket Chip processor cores using the Chipyard framework. The highly scalable processing cells are implemented as modified RocketTiles with local scratchpad memory and direct TileLink/DTIM interconnects between neighboring cells. The Chisel-based implementation allows the generation, validation, and evaluation of various processing cell structures by applying the different Chipyard flows: Rocket Chip RTL generation, Verilator RTL simulation, FireSim RTL simulation, FPGA synthesis, and the VLSI flow. The individual GPC grids were configured in Chisel through parameters from which Verilog RTL code was generated. The generated Verilog RTL was then used for Verilator simulation. The software was compiled and loaded via a frontend server (FESVR), which was modified to load the individual software binaries to the distributed local scratchpad memories of the processing cells. For an alternative RTL simulation, we also applied FireSim and synthesized our simulator for the Alevo U280 cards of the Noctua2 HPC cluster of Paderborn University. Comparisons of FireSim vs. Verilator gave speedups from 70.2 to 116.9 at effective target frequencies of 51.8 MHz. As such, FireSim turned out to be a very attractive high-performant alternative for RTL simulation. Current studies investigate their scaling to combine singles boards through optical links to much larger grids.

Our FPGA synthesis with Vivado applied different 2-dimensional grid configurations running at 100 MHz on various FPGA evaluation boards (*Arty7*, *VC707*, *VCU108*), which have shown the limits of *Arty7 100T* boards to 2×2 grids and motivates *VCU108* boards for larger grids. Our VLSI flow generated layouts for SkyWater 130 nm technology with Cadence Genus and Innovus. First studies indicated areas from 9 mm^2 for 2×2 grids up to 138 mm^2 for 8×8 grids with no signification impact of the communication structures in final chip layouts.

For the validation of the functional correctness of our approach in the context of the Chipyard framework, we started with several simple C programs to test the compilation and distribution of software binaries to the individual processing cells for different grid sizes in Verilator simulation and on FPGA. First more detailed performance studies investigated a scalable systolic matrix multiplication, which was tested on various sizes up to a 8×8 grid. The final comparison demonstrates that the hardware solution with CSR handshakes yields speedups ranging from 7% to 16% compared to the software solution with shared memories. However, further in-depth investigations are required with different applications and scalings before general conclusions can be drawn. Future work will further advance our parametrizable GPC RISC-V tiles towards ad-

ditional generic local communication structures and their combination with shared memory architectures.

Acknowledgements

The work described herein is partly funded by the German Bundesministerium für Bildung und Forschung (BMBF) through the Scale4Edge project (16ME0133).

References

[1] *AMBA® AXI-Stream Protocol Specification*. Arm Limited. Apr. 2021. URL: https://developer.arm.com/documentation/ihi0051/latest.

[2] 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.

[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] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[11] 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.

[12] Vivek Govindasamy and Rainer Dömer. "Instruction-Level Modeling and Evaluation of a Cache-Less Grid of Processing Cells". In: *Proceedings of Forum on Specification and Design Languages*. Turin, Italy, 2023.

[13] 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.

[14] 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.

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

[16] 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.

[17] Lars Luchterhandt et al. "Towards a Rocket Chip Based Implementation of the RISC-V GPC Architecture". In: *MBMV 2023 - 26. Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*. VDE Verlag, 2023.

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

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

[20] 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.

[21] Claudio Raccomandato, Emad Arasteh, and Rainer Dömer. "MapGL: Interactive Application Mapping and Profiling on a Grid of Processing Cells". In: *Proceedings (research track) of the Design and Verification Conference in Europe*. Munich, Germany, 2023.

[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] 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.

[28] 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.

[29] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Tech. rep. EECS Department, University of California, Berkeley, Dec. 2019.

[30] Andrew Waterman et al. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9*. Tech. rep. UCB/EECS-2016-129. EECS Department, University of California, Berkeley, July 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html.

[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.