embeddedworld2020
Exhibition&Conference
... it's a smarter world

The ParaNut/RISC-V Processor - An Open, Parallel, and Highly Scalable Processor Architecture for FPGA-based Systems

Alexander Bahle, Gundolf Kiefer
Efficient Embedded Systems Group
Augsburg University of Applied Sciences
Augsburg, Germany
{alexander.bahle, gundolf.kiefer}@hs-augsburg.de

Anna Kerstin Pfützner, Lutz Vollbracht
IBV - Echtzeit- und Embedded GmbH & Co. KG
Augsburg, Germany
annakerstin.pfuetzner@gmail.com,
vollbracht@ibv-augsburg.net

***Abstract*—The paper presents a customizable, highly scalable, and RISC-V compatible processor architecture for FPGA-based systems. A key aspect of the *ParaNut* architecture is a special concept of parallelism, which combines advantages of SIMD vectorization and simultaneous multi-threading in one architecture. At the same time, the complexity of a single computing core is minimized in order to save area and power. Speculation techniques are generally avoided in order to save power and to make the processor robust against security flaws.**

The design is presently used in education and research. In addition, the present implementation passes the RISC-V compliance tests (RV32IM instructions) and is thus compatible with the standard RISC-V toolchain. Preliminary experiments on a Xilinx 7 platform show 0.87 CoreMarks/MHz using 1 core and 3.45 CoreMarks/MHz using 4 cores, revealing an almost perfect speedup of 3.97.

***Keywords*—RISC-V, Processor Architecture, FPGA, SIMD**

I. Introduction

General-purpose processors are used in virtually any embedded system because they are easily software-programmable and have mature toolchains. This great development support results in a small time to market which helps to keep prices down for producers as well as consumers. Rising requirements and demand for low power devices in Internet of Things (IoT) applications lead to application-specific hardware with special processors used for artificial intelligence [1] and computer vision [2]. This trend and the upcoming end of Moore's Law drive the development of more efficient systems, but with customized hardware.

Customizable processors offer the advantages of general-purpose CPUs with the option to optimize the hardware for special tasks, which, in turn, allows highly specialized system architectures at relatively low expense. Decreasing cost of field-programmable gate arrays (FPGAs) enables the implementation of application-specific hardware with highly customizable soft-core processors even at small quantities. Furthermore, the growing capacity and efficiency of FPGAs allow to implement Systems-on-a-Chip (SoCs) containing both specialized hardware together with general-purpose processors running increasingly complex software. To increase the performance of software, one can add more cores to leverage the principles of multithreading or add a form of single instruction multiple data (SIMD) hardware. Graphics processing units (GPUs) or special SIMD extensions for CPUs are common examples for SIMD hardware. However, in order to use them efficiently deep knowledge about the system architecture and special instructions, such as *Intel AVX* or *ARM NEON*, are needed. Leveraging a GPU typically requires to (re)write the software using *OpenCL* or *CUDA*.

This paper presents a customizable and highly scalable implementation of the *ParaNut* processor architecture using the RISC-V instruction set architecture (ISA). The purpose of the *ParaNut* architecture is to bridge the gap between thread-level and data-level parallelism by providing a simple programming model. In particular, special SIMD instructions are avoided. The *ParaNut* specific features are abstracted through RISC-V standard compatible control and status registers (CSRs). This allows the use of standard ISA compatible toolchains (GCC) for software development. From an economic point of view, the *ParaNut* processor with its features may be one step on a path to a new kind of "deeply engineered" systems that can combine high efficiency and low cost even for smaller quantities.

Section II gives an overview of other customizable processors for FPGA computing platforms as well as other SIMD extensions, such as the RISC-V vector extension. The basics of the *ParaNut* architecture and design considerations that go along with it are outlined in section III. Section IV describes the special concepts of parallelism present in the architecture. Status of compliance with the RISC-V standards and the current state of software development support is presented in

section V. Experiments and their results are laid out in section VI, followed by section VII, which concludes the paper.

II. Related Work

Customizable soft-core processors offer different levels of configurability and thus can be tailored to specific demands. Examples of commercial soft-core processors are the *Xilinx MicroBlaze* [8] and *Altera/Intel Nios II* [9] which are only usable on the vendors' FPGAs. Open source soft-cores are less restrictive towards specific devices due to the availability of the hardware description language (HDL) source code. Long running projects and cores are *Cobham Gaisler LEON2/3/4* [10], *OpenRISC* [11], or *Sun OpenSPARC* [12]. Since the release of the RISC-V ISA a new wave of open-source processors has been developed for various application fields. On a 32-bit microcontroller scale, some examples are the *Z-scale* [13], *PULPino/PULPissimo* [14], *PicoRV32* [15], or the *Freedom E310* [16].

Several general-purpose processors feature extensions with special instructions for SIMD vectorization. For ARM cores the vector extension is called *VFPv3/NEON* [17], for OpenRISC *ORVDX64* [11], for SPARC *VIS* [18] and for RISC-V *RISC-V Vector Extension* [19]. Taking the latest stable draft of the *RISC-V Vector Extension* as an example, one can see that it requires a set of control and status registers (CSRs) and a special set of vector instructions to execute vectorized software. The support for vector operations is currently not implemented in the upstream GCC toolchain, and none of the aforementioned small RISC-V soft-core processors presently supports the vector extension.

III. Basic Architecture and Design Considerations

The basic design considerations and hardware architecture of the *ParaNut* processor are described in detail in [4] for an earlier *OpenRISC* variant of the processor. This and the next section summarize the most relevant aspects in the context of the new *RISC-V* version.

A. General Design Goals

Techniques for exploiting parallelism in processors can be grouped into three categories, namely *data-level parallelism (DLP)*, *instruction-level parallelism (ILP)*, and *thread-level parallelism (TLP)* [3]. The ILP category includes numerous techniques that aim at improving the single-core performance, such as (deep) pipelining, out-of-order execution, or VLIW (very long instruction word) processing. ILP techniques generally come at the expense of high area usage and power consumption, which typically increase at a higher rate than performance improvements. Also, ILP often involves speculation techniques, which may be undesired from a security point of view. For example, "Spectre" and "Meltdown" are attacks that exploit speculative and out-of-order execution in some way [21], [22]. For these reasons, the *ParaNut* design uses ILP very conservatively and focuses on data-level and thread-level instead.

In summary, the design of the *ParaNut* architecture was guided by the following considerations:

- Optimize the architecture of the cores for area, not speed.
- Provide an efficient memory/cache subsystem to support many parallel processing units.
- Provide an SIMD execution model to support SIMD vectorization using standard instructions and high-level languages.

B. Processor Structure

Figure 1 shows a block diagram of an exemplary *ParaNut* instantiation with four full-featured cores (alternatives will be explained in Section IV). Each core contains an ALU, a register file, and some control logic, which together form the *Execution Unit (ExU)*. The instruction port (*IPort*) is responsible for fetching instructions from the memory subsystem and contains a small buffer for prefetching instructions. The data port (*DPort*) is responsible for performing the data memory accesses of load and store operations. It contains a small store buffer and implements write combining and store forwarding mechanisms as well as mechanisms to support atomic operations.

The *Execution Unit* is designed and optimized for a best-case throughput of one instruction in two clock cycles (CPI = 2, CPI = "clocks per instruction"). Unlike other pipeline designs targeting a best-case CPI value of 1, this allows to better optimize the execution unit for area, since no pipeline registers or extra components for the detection and resolution of pipeline conflicts are required. In addition, several measures in the *ParaNut* design help to maintain an average-case throughput very close to the best-case value of CPI = 2, even for multi-core implementations (for example, buffers in the *IPorts* and *DPorts*).

The *Memory Unit (MemU)* contains the cache, the system bus interface, and a multitude of read and write ports for the processor cores. Each core is connected to the *MemU* by two independent read ports for instructions and data and one write port for data. All transactions to and from the system bus (AXI or Wishbone) are handled by a single bus interface unit. The cache logically operates as a shared cache for all cores. It can be configured to be 1/2/4-way set associative with a configurable replacement strategy (e.g. least-recently-used or pseudo-random).

The design of the memory interface and cache organization is very critical for the scalability of many-core systems. This is attributed to several measures inside the *Memory Unit*. The cache is organized in independent banks with switchable paths from each bank to each read and write port. Cache tag data is replicated to allow arbitrary concurrent lookups. Parallel cache data access by different ports can be performed concurrently if their addresses a) map to different banks or b) map to the same memory word in the same bank. In practice, most concurrent memory accesses fall into one of these categories. Moreover, by using dual-ported Block-RAM cells, each bank can be equipped with two ports, so that up to two conflicting accesses (i. e. same bank, different addresses) are possible in

parallel. The likelihood of contention (i. e. memory accesses that cannot be handled in parallel) can be reduced at will by increasing the number of banks which can be configured at synthesis time.

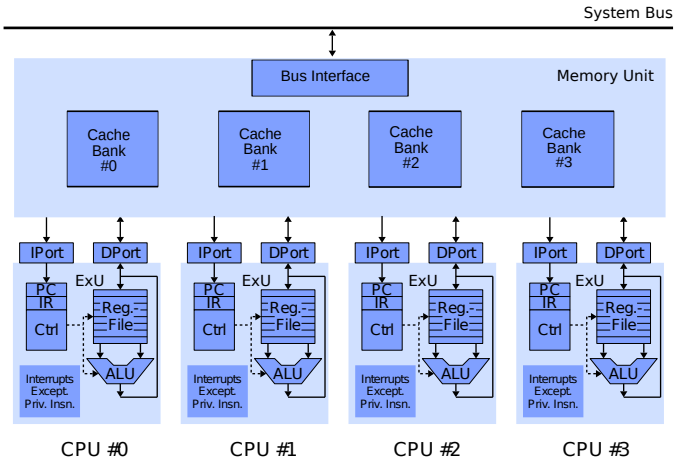

Fig. 1: Block diagram of a *ParaNut* with 4 (full-featured) cores

C. Design Methodology

The ParaNut hardware is modeled completely in SystemC and in general, the same code model is used for hardware synthesis as well as for building a cycle-accurate instruction set simulator. This ensures that the simulator reflects the real hardware behaviour.

The simulator supports the development and debugging of both hardware and software through the ability to produce VCD trace files to inspect the inner workings of the processor and through an OpenOCD-compatible remote-bitbang (RBB) interface.

For hardware synthesis, major parts of the SystemC processor model conform to the *SystemC Synthesizable Subset* [20]. Only some performance-critical modules are implemented in VHDL. To date, the implementation is mainly used in education and research. Hence, the implementation is not yet fully optimized at the gate level.

IV. The ParaNut Architecture: Concepts of Parallelism

A. Overview

The *ParaNut* architecture puts a strong emphasis on parallelism on data-level (i.e. SIMD vectorization) and on thread-level. Common CPU vector SIMD extensions have serious restrictions that make them difficult to support by compilers, for example:

- fixed vector widths and available data types,
- inefficient conditional execution with individual vector elements,
- limited memory addressing modes (e.g. vectors can only be loaded from subsequent addresses, ”scatter”/”gather”-like accesses are not possible).

A practical example for the latter case is the calculation of histograms in image processing (see Figure 4(a)). In the body of the critical loop

```
for (int n = 0; n < DATA_SIZE; n++)
   histogram[(data[n] & bitmask) >> shifts]++;
```

the array *histogram* is accessed in a way that the array index is calculated individually for each n and depends on the input array *data*. Therefore the effective memory address changes randomly from iteration to iteration and the *for* loop cannot be vectorized with the common SIMD vector extensions of most CPUs.

On the other hand, common SIMD extensions often offer specialized instructions like ”add with saturation”. These can be useful in signal or image processing applications, but are very difficult to support by compilers due to their complex semantics. This makes them more or less useless for programming in a high-level language.

For this reason, the *ParaNut* architecture aims to support SIMD vectorization based on just the standard CPU instruction set. It introduces a programming model in which the complete software can be written in a standard programming language so that a paradigm can be used for SIMD vectorization that is very similar to thread-level parallelism

B. Linking Cores for SIMD Vectorization

In the *ParaNut linked mode* multiple CPU cores are linked together as sketched in Figure 2: The data paths, comprised of the ALUs, register files, and memory data ports, operate individually and independently of each other. Only one instance of the execution control hardware (comprised of the program counter, instruction unit, the instruction memory port, and some more logic) is active, and all cores are controlled by the same control signals originating from the master CPU (#0 in Figure 2). As a result, all cores now execute identical instructions with individual data in a synchronous way and the hardware behaves like a SIMD processor. However, unlike the instructions in special SIMD vector extensions, the commands are standard RISC-V instructions.

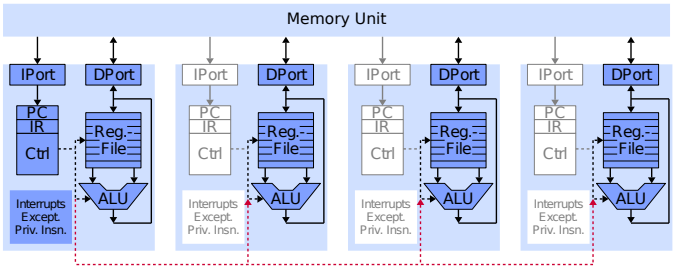

Fig. 2: *ParaNut* processor operating in *linked mode*

From a software perspective, this behaviour is almost equivalent to that of a multi-core processor running the same code in multiple independent threads. The operation is fully equivalent as long as the instruction sequence does not contain any conditional branches or indirect jump instructions that evaluate

differently on the cores. Considering this restriction, SIMD vectorization can be adopted using standard instructions.

For example, the C code fragment

```
int n, a[4], b[4], w[4], wsum[4];
...
for (n = 0; n < 4; n++)
  wsum = a[n] * w[n] + b[n] * (100 - w[n]);
...
```

can be transformed and vectorized as follows:

```
int n, a[4], b[4], w[4], wsum[4];
...
n = pn_begin_linked (4);
/* turns on linked mode, returns core id (0..3) */
wsum = a[n] * w[n] + b[n] * (100 - w[n]);
pn_end_linked ();
/* switches back to single-thread mode */
...
```

Initially, the processor runs in single-thread mode, with the primary core executing code and the linked cores being inactive. The macro *pn_begin_linked()* activates the linked cores with the effect that the subsequent code is executed synchronously 4 times in parallel. Conversely, the macro *pn_end_linked()* marks the end of the parallel section and causes the linked cores to be deactivated again. Another example is shown below in Figures 3 (a) and 3 (b).

From a hardware perspective, the *linked mode* has considerable advantages over full-featured multi-processing: Linked cores do not emit any memory accesses for instruction fetches. This may greatly reduce the load on the memory subsystem and avoids system bus contention. Furthermore, a considerable amount of hardware can be saved by omitting the control logic of cores that are only operated in *linked mode*.

C. Thread-Level Parallelism

The *ParaNut* architecture also allows the definition of cores that can execute independent threads, but are lacking some features such as support for interrupts, exceptions, and privileged system instructions in order to save hardware. The programming interface (see Section V-B) is very similar to the *linked mode* interface and contains two macros *pn_begin_threaded()* and *pn_end_threaded()* to open and close a parallel section for conventional multi-threaded code.

As mentioned in Section IV-B, in linked mode certain restrictions related to conditional branches apply, which limits the use of ”if” and ”case” instructions, for instance. In thread mode there are no restrictions on used instructions. The transition from linked mode to thread mode code is facilitated by the thread mode programming interface and vice versa. Figure 3 shows an example of a loop (Figure 3a) and its vectorized variants in linked mode (Figure 3b) and thread mode (Figure 3c) respectively.

For (multi-core) processors that do not support the linked mode, the linked mode macros can be mapped to their thread mode counterparts, and the code will still execute correctly in multiple parallel threads. Therefore, source code containing sections for the *ParaNut's* linked mode still remains fully portable.

```
int a[1000], b[1000], s[1000];
int n, id;
...
for (n = 0; n < 1000; n += 1)
  s[n] = a[n] + b[n];
```

(a) Original (sequential) version

```
int a[1000], b[1000], s[1000];
int n, id;
...
id = pn_begin_linked (4);
for (n = 0; n < 1000; n += 4)
  // Note: n is always identical in all threads
  s[n + id] = a[n + id] + b[n + id];
pn_end_linked ();
```

(b) Vectorized version (Linked Mode)

```
int a[1000], b[1000], s[1000];
int n, id;
...
id = pn_begin_threaded (4);
for (n = 0; n < 1000; n += 4)
  // Note: n is always identical in all threads
  s[n + id] = a[n + id] + b[n + id];
pn_end_threaded ();
```

(c) Threaded version (Thread Mode)

Fig. 3: Parallel loop example demonstrating the usage of (b) *linked mode* and (c) *thread mode*

```
uint32_t *histogram, *data, bitmask, shifts;
...
for (int n = 0; n < DATA_SIZE; n++)
   histogram[(data[n] & bitmask) >> shifts]++;
...
```

(a) Original (sequential) version

```
uint32_t *histogram, *data, bitmask, shifts,
chunk, cpuid;
...
cpuid = pn_begin_linked(4);
/* Each core gets its own result array */
histogram = histogram + HIST_SIZE*cpuid;
for (int n = cpuid; n < DATA_SIZE; n += 4)
    histogram[(data[n] & bitmask) >> shifts]++;
pn_end_linked();
/* Reduce all results into one histogram */
...
```

(b) Vectorized version (Linked Mode)

```
uint32_t *histogram, *data, bitmask, shifts,
       chunk, cpuid;
...
cpuid = pn_begin_threaded(4);
/* Each core gets its own result array */
histogram = histogram + HIST_SIZE*cpuid;
for (int n = cpuid; n < DATA_SIZE; n += 4)
       histogram[(data[n] & bitmask) >> shifts]++;
pn_end_threaded();
/* Reduce all results into one histogram */
...
```

(c) Threaded version (Thread Mode)

Fig. 4: Basic histogram calculation (a) and the parallel code for the *linked mode* (b) and *thread mode* (c)

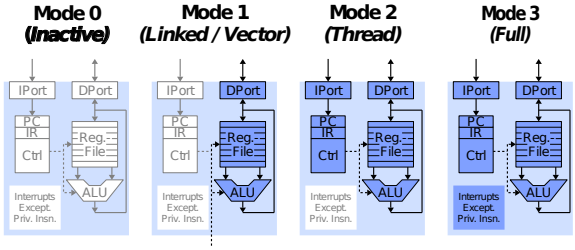

Fig. 5: Modes of a *ParaNut* core

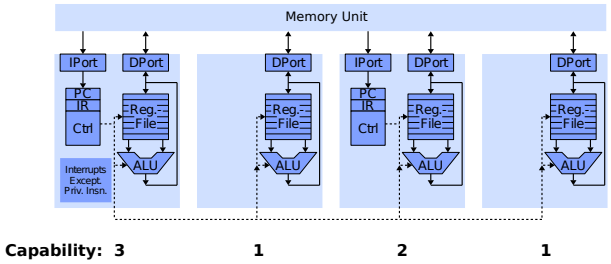

Fig. 6: Example of a *ParaNut* instantiation with cores of different capabilities

D. Modes and Capabilities

Based on the different modes of operation discussed so far, the *ParaNut* architecture defines 4 different *CPU modes*, which are sketched in Figure 5. In mode 0, the core is inactive (i. e. currently unused). In mode 1, the core operates in the linked mode as explained above. A core running in mode 2 can execute a thread autonomously. It supports all standard instructions, but no interrupts, exceptions, or privileged system instructions. Finally, a processor in mode 3 represents a full-featured CPU. Usually, only one core needs to support mode 3, which allows a considerable amount of complexity to be saved for the other cores while maintaining all options for efficient thread-level or data-level parallelism.

It is not necessary that each core supports all 4 modes. At synthesis time, the available modes can be selected on a per-core basis by means of a *capability level*. A capability level of n means that the core can operate in mode n or any mode below at runtime. For example, Figure 6 shows a *ParaNut* instance with one capability-3, one capability-2, and two capability-1 cores. This processor can be arbitrarily configured at runtime to execute, for example, two threads in parallel or one thread with 4-way SIMD-vectorized code (see Figure 7).

V. RISC-V Compliance and Software Development Support

A. RISC-V Compliance and Debugging

The *ParaNut/RISC-V* implementation aims to be fully compatible with standard RISC-V compilers and tools. It is based on the *RISC-V Instruction Set Manual Volume I and Volume II Version 20190608* [23], [24]. For support with the standard debug tools the RISC-V *External Debug Support Version 0.13* was used to implement a *Debug Module* [25].

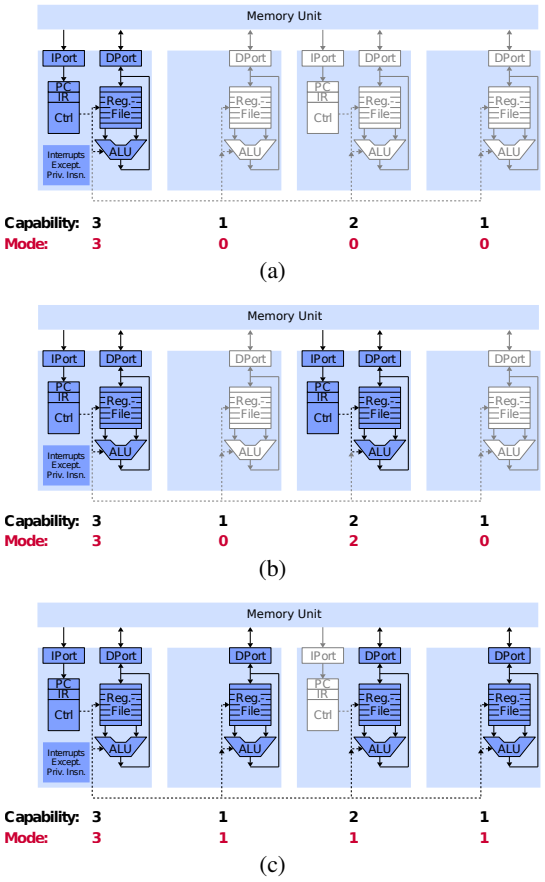

Fig. 7: *ParaNut* of Figure 6 running (a) single-threaded code, (b) two independent threads, and (c) 4-way vectorized SIMD code

The *RISC-V Foundation Compliance Task Group* provides a repository containing multiple compliance tests [26]. By providing a user-defined target for the tests, the *ParaNut/RISC-V* SystemC model has been validated to conform to the *RV32I Base Integer Instruction Set* and the *RV32M Standard Extension for Integer Multiplication and Division* standards. The *ParaNut/RISC-V* also supports a subset of the *RV32A Standard Extension for Atomic Instructions*, namely the "lr.w" and "sc.w" instructions, which are sufficient to implement all standard synchronization mechanisms (e.g. mutex, semaphore). Table I shows an overview of all supported RISC-V extensions. The third column contains a "✓" if the compliance test for the extension in question has been successfully passed, and the fourth column shows whether the extension is optional at synthesis time.

TABLE I: Supported RISC-V Extensions

Name	Description	Com.	Opt.
RV32I	Base Integer Instruction Set	✓	
RV32Zicsr	Control and Status Registers	✓	
RV32M	Integer Multiplication and Division	✓	✓
RV32A	Atomic Instructions	(see text)	✓

B. Software Support Library

In order to support the *ParaNut*-specific hardware features, a support library *libparanut* has been developed. The aim of this C library is to encapsulate all features not common to RISC-V in a portable and lightweight way.

libparanut provides a portable application programming interface (API) for the following functionalities:

Basics:
This includes some basic functions, for example reading the processor status or the configuration.

Linked Mode:
A set of functions for turning the linked mode on and off, either by a passed number of cores or by a core bitmask. Central functions include:

```
PN_CID pn_begin_linked(PN_NUMC numcores);
PN_CID pn_begin_linked_m(PN_CMSK coremask);
PN_CID pn_begin_linked_gm(
          PN_CMSK *coremask_array,
          PN_NUMG array_size);
int    pn_end_linked(void);
```

Thread Mode:
A set of functions for turning the thread mode on and off. The API is very similar to the linked mode functions so that application code that is no longer suitable for linked mode execution can easily be changed to operate in thread mode. Central functions include:

```
PN_CID pn_begin_threaded(PN_NUMC numcores);
PN_CID pn_begin_threaded_m(PN_CMSK coremask);
PN_CID pn_begin_threaded_gm(
          PN_CMSK *coremask_array,
          PN_NUMG array_size);
int    pn_end_threaded(void);
```

Synchronization:
Set of functions to provide synchronization between parallel *ParaNut* cores. At present, a spinlock is implemented. Further synchronization primitives may be added in future versions.

Memory Unit:
Functions for managing the memory unit, such as cache invalidation or cache flushing.

Exceptions and Interrupts:
Handlers for *ParaNut*-specific exceptions and interrupts and support functions to let the application programmer define their own handlers.

The *libparanut* library is compilable using a standard RISC-V GCC toolchain and is written in a generic way to support arbitrary *ParaNut* configurations without recompilation. Currently it is used together with the RISC-V *newlib* 2.5.0 port. A Linux user-mode variant of *libparanut* is planned to support future processor versions running a Linux operating system.

libparanut is provided with an extensive unit test, and the test cases serve as code examples for the correct API usage in addition to the existing documentation.

VI. Experimental Results

A. Setup

The *ParaNut/RISC-V* processor was configured according to table II. A *ZYBO Z7-20 Zynq-7000 ARM/FPGA SoC platform* featuring a Xilinx XC7Z020-1CLG400C device was used to host the system. Figure 8 shows an overview of the system. The ARM Application Processing Unit (APU) present on the SoC is used to handle serial UART communication to the host machine and to load software into the DDR3 memory.

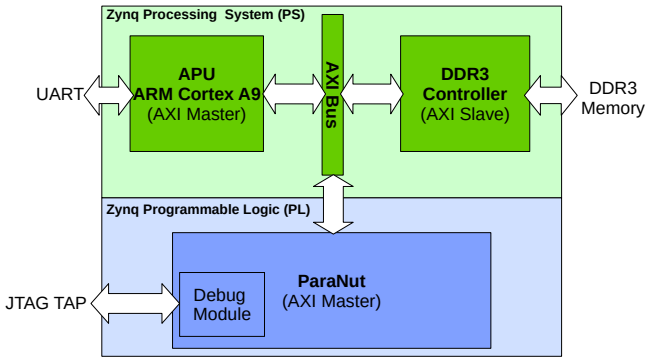

Fig. 8: Block diagram of the *ParaNut/RISC-V* system

TABLE II: *ParaNut/RISC-V* Benchmark configuration

Parameter	Value
Clock Speed	20 MHz
CPU Cores	1...8
M-Extension	✓
A-Extension (lr.w and sc.w)	✓
Cache size	32 kB
Cache sets	512
Cache line size	16 Bytes (4 Banks)
Cache associativity	4 ways
Cache replacement strategy	LRU
Instruction buffer size (IPort)	4 words
Write buffer size (DPort)	4 words

B. Synthesis Results

The different configurations of the *ParaNut/RISC-V* have been synthesized using *Xilinx Vivado 2017.2*. Table III shows the slice usage of the *ParaNut/RISC-V* featuring 1 core with a capability of 3 and up to 7 cores with a capability of 2. For comparison, a *RocketChip* based *Freedom E310* with standard options was synthesized for the *Artix-7 35T Arty FGPA Evaluation Kit*. The last row shows the size of a single processor "tile" including the core itself and the instruction/data cache of 16kB each. As mentioned above, the *ParaNut/RISC-V* is not fully optimized yet to facilitate its use in education and research, and the single-core *ParaNut* uses slightly more slices than the *Freedom E310*. However, a multi-core system would require a complete "tile" per core, whereas multiple cores of a *ParaNut* can share parts of the memory logic.

Figure 9 shows the distribution of slices over the main components of the *ParaNut* processor. The sublinear growth in overall size is a big advantage over other cores that need to add a whole new core to the system to execute multiple

threads. Figure 10 shows the effect the maximum capability can have on resource usage. The mode 1 capable cores do not require their own IPort and therefore an additional port to the MemU. This, in turn, frees up 2094 slices in this case, while retaining the ability to run data-parallel software.

TABLE III: Zynq 7000 resource usage

Cores	Slice LUTs	Slice FFs	Slices	Increase
1	7,139	3,759	2,504	1.00
2	12,433	6,094	4,185	1.67
4	23,599	10,712	7,265	2.90
8	44,393	19,536	12,473	4.98
Freedom E310 (1 tile)	6,713	4,131	2,139	

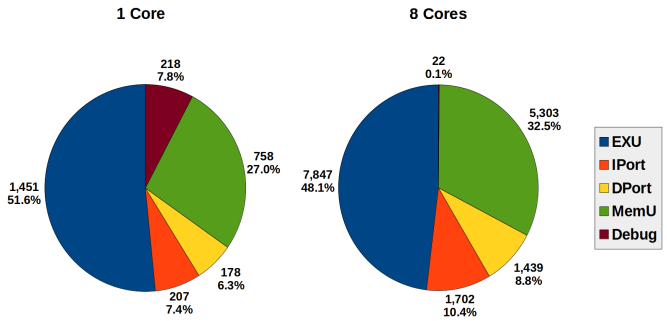

Fig. 9: Distribution of slices over main components of the *ParaNut* processor

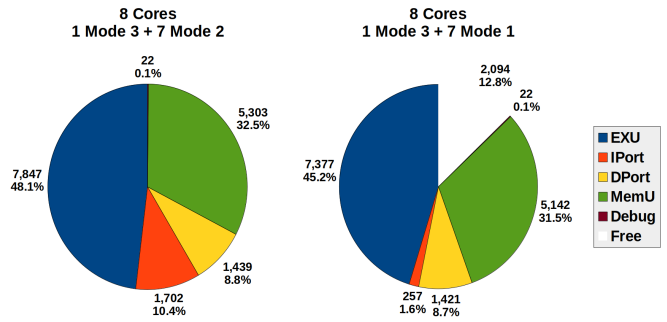

Fig. 10: Distribution of slices over main components of a *ParaNut* processor using 8 cores with different capabilities

C. Benchmark Results

The *CoreMark* benchmark was compiled using GCC version 8.3.0 and the "newlib" C library. Compiler options were set according to the hardware configuration (-O3 -march=rv32im -mabi=ilp32). The benchmarks were evaluated at a clock frequency of 20 MHz.

Table IV shows the results of the *CoreMark* benchmark run, which was performed several times with an increasing number of parallel threads. The results show that although the *ParaNut/RISC-V* has a shared cache for its cores, at 4 cores an almost perfect speedup of 4 is achieved, and even at 8 cores the speedup is about 7.6.

To evaluate the performance of the new linked mode (mode 1) the histogram calculation program shown in Section IV-B and Figure 4 has been executed on a system with 4 cores for a data size of $1920*1080*4$ $Bytes$. The runtime was measured using the internal timer and the results are shown in table V.

TABLE IV: CoreMark benchmark results

Processor	Cores	CoreMark/MHz	Speedup
ParaNut	1	0.87	1.00
	2	1.73	2.00
	4	3.45	3.97
	8	6.59	7.59
MicroBlaze [28]	1	1.90	
HiFive Unleashed [28]	1	2.01	

TABLE V: Histogram runtime results

Type	Time/ms	Speedup
Sequential (a)	4,616.03	1.00
Linked Mode (b)	1,467.11	3.14
Thread Mode (c)	1,437.55	3.21

D. Demonstrator System "Pong-on-a-Chip"

In order to evaluate the practical usability of the processor architecture and the support library *libparanut*, a Pong-like mixed reality game has been implemented. The system incorporates a *ParaNut* processor as the main processor and a hardware-accelerated image processing chain using the *ASTERICS* framework [29]. It is implemented on the same *ZYBO Z7-20* FPGA board as used for the previous experiments.

Figure 11 shows the application. A virtual red ball is drawn over a camera image and animated in such a way that the ball is reflected by edges detected in the camera image. Players can put their hands or objects in front of the camera and play with the ball.

The *ASTERICS* subsystem contains an edge detector implemented in hardware, while the complete software and game application are running on a *ParaNut*. It should be noted that the software includes a complex software library for the *ASTERICS* subsystem, including drivers for the camera and image processing chain as well as a drawing library for the overlay, which has been executed on *ARM* processors before.

VII. Conclusion

The *ParaNut/RISC-V* processor is a new, open, scalable, and RISC-V compatible processor based on the *ParaNut* architecture. The special concepts of different execution modes and capabilities allow for thread and data-level parallelism. The implementation is available as a SystemC model, serving both as an instruction set simulator and as the basis for an FPGA-proven SystemC/VHDL synthesis model. Easy access and abstraction to the *ParaNut*-specific features, especially to control the linked and threaded execution, are provided by the *libparanut* software library. Benchmarks show promising results with respect to scalability at 4 and 8 cores.

The project is still actively in development and leaves room for optimizations and improvements. Ongoing work focuses on improving the timing and overall performance. Furthermore, the development of a memory management unit (MMU) has already been started to support operating systems like Linux in the future.

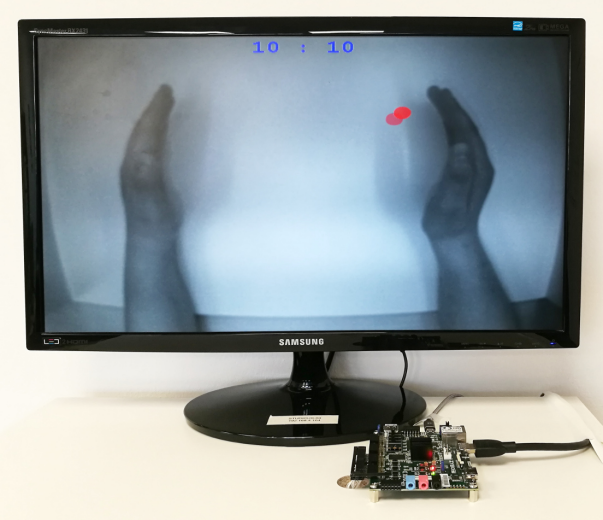

Fig. 11: Pong-like mixed reality game with *ASTERICS* and a *ParaNut* processor

Acknowledgment

The authors would like to thank Walter Eberl-Schell, Christian Merkle, Christian Meyer, Dominic Rath, Michael Schäferling, and Michael Seider for their valuable comments and suggestions for improving this work.

References

[1] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson, "Motivation for and evaluation of the first tensor processing unit." IEEE Micro 38.3 (2018): 10-19, May 2018
[2] Gideon P. Stein, Elchanan Rushinek, Gaby Hayun, and Amnon Shashua, "A computer vision system on a chip: a case study from the automotive domain.", 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)-Workshops, September 2005
[3] John L. Hennessy and David A. Patterson, "Computer Architecture - A Quantitative Approach" (6th Edition), Morgan Kaufmann, 2017, ISBN 9780128119051
[4] Gundolf Kiefer, Michael Seider, and Michael Schäferling, "ParaNut - An Open, Scalable, and Highly Parallel Processor Architecture for FPGA-based Systems", Embedded World Conference, Nürnberg, February 2015
[5] Xilinx Inc., "DS190: Zynq-7000 SoC Data Sheet: Overview", July 2018
[6] Xilinx Inc., "DS890: UltraScale Architecture and Product Data Sheet: Overview", August 2019.
[7] Intel Co., "Intel Stratix 10 Hard Processor System Technical Reference Manual", May 2019
[8] Xilinx Inc., "UG081: MicroBlaze Processor Reference Guide", January 2008
[9] Intel Co., "NII5V1: Nios II Classic Processor Reference Guide", June 2016
[10] Gaisler Jiri and Isomäki Marko, "LEON3 GR-XC3S-1500 Template Design.", Gaisler Research, 2006.
[11] opencores.org, "OpenRISC 1000 Architecture Manual, Architecture Version 1.0", December 2012
[12] Sun Microsystems Inc., "OpenSPARC T2 Core Microarchitecture Specification", December 2007
[13] Yunsup Lee, Albert Ou, and Albert Magyar, "Z-scale: Tiny 32-bit RISC-V systems.", Open-RISC Conf. Geneva, Switzerland, June 2015
[14] Traber Andreas, et al., "PULPino: A small single-core RISC-V SoC.", 3rd RISCV Workshop, January 2016.
[15] Wolf Clifford, "PicoRV32 - A Size-Optimized RISC-V CPU", 2019, [Online]. Available: https://github.com/cliffordwolf/picorv32
[16] SiFive, "SiFive's Freedom platform", [Online], Available: https://github.com/sifive/freedom
[17] ARM Limited, "Cortex-A9 NEON Media Processing Engine", Revision: r2p2, April 2010
[18] Wozniak Andrzej, "Using video-oriented instructions to speed up sequence comparison.", Bioinformatics 13.2: 145-150, 1997
[19] Alon Amid, et al., , "RISC-V Vector Extension, Version 0.8", December 2019
[20] Accellera Systems Initiative Inc., "SystemC Synthesizable Subset Version 1.4 Draft", January 2015
[21] Kocher, Paul, et al., "Spectre attacks: Exploiting speculative execution.", 2019 IEEE Symposium on Security and Privacy (SP). IEEE, May 2019
[22] Lipp, Moritz, et al., "Meltdown.", arXiv preprint arXiv:1801.01207, January 2018
[23] Andrew Waterman and Krste Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified", RISC-V Foundation, June 2019
[24] Andrew Waterman and Krste Asanović, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified", RISC-V Foundation, June 2019
[25] Tim Newsome and Megan Wachs, "RISC-V External Debug Support Version 0.13", RISC-V Debug task group, October 2018
[26] RISC-V Compliance Task Group, [Online], Available: https://github.com/riscv/riscv-compliance
[27] Krste Asanović, et al., "The Rocket Chip Generator", Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, April 2016
[28] Embedded Microprocessor Benchmark Consortium (EEMBC), "CoreMark Benchmark Scores Database.", December 2019. [Online]. Available: https://www.eembc.org/coremark/scores.php
[29] Efficient Embedded Systems Group UAS Augsburg, [Online], Available: https://ees.hs-augsburg.de/asterics