

Atalanta: Open-Source RISC-V Microcontroller for Rust-Based Hard Real-Time Systems

Antti Nurmi¹[0000-0003-3533-9832], Per Lindgren^{1,2},
Abdesattar Kalache¹[0000-0002-5560-3250],
Henri Lunnikivi¹[0000-0003-4817-2939], and
Timo D. Hämäläinen¹[0000-0002-7867-0800]

¹ Tampere University, Tampere, Finland

² Luleå University of Technology, Luleå, Sweden

{antti.nurmi, abdesattar.kalache, henri.lunnikivi,
timo.hamalainen}@tuni.fi
per.lindgren@ltu.se

Abstract. Real-time systems are a segment of embedded systems that have remained dominated by proprietary hardware architectures, despite the continuing growth of the open-source RISC-V instruction set architecture (ISA). The introduction of core-local interrupt controller (CLIC) extensions to the RISC-V architecture presents a promising opportunity to bridge the technological gap with ARM in low-latency interrupt handling. Regarding software, the real-time interrupt-driven concurrency (RTIC) framework enables ever lighter hard real-time systems with formal compile-time guarantees for memory safety, response time and overall schedulability.

In this publication we adapt *Ibex*, a small, open-source RISC-V processor for CLIC support and present *Atalanta*, a lightweight microcontroller designed around the RTIC framework. *Atalanta* implements a localized memory architecture that enables low-latency context switching together with a large number of supported interrupt inputs and levels provided by the CLIC specification.

We evaluate *Atalanta* for real-time performance and implementation feasibility through simulation-based measurements and FPGA prototyping, respectively. We are able to demonstrate an interrupt latency of 5 cycles with minimal jitter and a context switch latency of 21 cycles, placing it competitively against current state-of-the-art solutions. Furthermore, we implement an FPGA prototype for the Xilinx PYNQ-Z1 and VCU118 boards, targeting a frequency of 45 MHz. We publish the sources and implementation scripts of *Atalanta* under a permissive open-source license.

Keywords: RISC-V · RTIC · CLIC · FPGA

1 Introduction

Real-time systems are an integral part of many mission- and safety-critical systems across a number of application domains, such as control units in robotics, aviation or automotive industries.

With growing demands on embedded hardware, the use of operating systems as an abstraction layer between the physical hardware and the application software is becoming ever more common. This abstraction comes with performance overhead that has been proven to be problematic when determining the worst-case execution time (WCET) characteristics for real-time systems. Reghenzani et al. [1] state that Linux, the most commonly used general-purpose operating system (GPOS), remains fundamentally unsuitable to meet hard real-time requirements in safety-critical systems despite use of the `PREEMPT_RT` patch.

Real-time operating systems (RTOS) are a specialized type of operating systems that are designed to better meet the needs of real-time systems. For a hard real-time system, a failure to meet a timing requirement, e.g, a missed deadline, is considered an error, whereas a soft real-time system continues to operate, albeit with a lower performance or quality of service. In the latter category we find many popular light weight operating systems such as FreeRTOS, ChibiOS, and Zephyr performing best-effort scheduling. Operating systems claiming hard real-time guarantees are less common and adopted mostly in safety critical domains, e.g, ThreadX and OSEK. However, even using a hard RTOS, it is still up to the developer to ensure that the system meets its timing requirements. Furthermore, it has been argued that thread-based programming models are fundamentally flawed for deterministic applications [2].

In this paper, we adopt the Rust-based Real-Time Interrupt-driven Concurrency (RTIC) framework [3], providing an executable task/resource application model. RTIC adheres to the stack resource policy (SRP) [4] model, and can be used to formally guarantee system-wide timing properties. Recent studies have shown that unsafe memory accesses are the root cause of the majority (70 %) of known security issues [5]. RTIC leverages on the Rust language and compiler to ensure memory safety. RTIC extends on Rust, providing the means for safe and deadlock-free concurrent access to shared resources. Moreover, an RTIC application never risks running into undefined behavior, unless explicitly defined as `unsafe` by the user. This way, RTIC provides a unique, correct by construction outset for implementing safety and security critical applications.

Lightweight real-time systems have traditionally been designed around proprietary ARM processor architectures. RISC-V [6] is an open-source instruction set architecture (ISA) that has been widely adapted in academia for research and education, as well as in a growing number of industry applications. However, real-time systems have remained dominated by ARM architectures, in part due to the relative immaturity of interrupt handling schemes on RISC-V [7]. More recently, works such as [8] and [9] have attempted to address this technological gap using the new core-local interrupt controller (CLIC) privileged architecture extensions [10] for the RISC-V ISA.

This work introduces, to the best of our knowledge, the first microcontroller platform designed specifically targeting the RTIC framework, which allows for a minimal, low-overhead hardware architecture that aims to be competitive with the ARM Cortex[®]-series. Moreover, this publication presents the following contributions:

- We extend the open-source *Ibex* CPU core [11] with a configurable interrupt interface and interrupt-level-based preemption to create the CLIC-conformant *RT-Ibex*. We present a characterization of *RT-Ibex* for performance and area requirement. We make the implementation available under a permissive open-source license³.
- We present *Atalanta*, a lightweight open-source⁴ microcontroller featuring the *RT-Ibex* and CLIC, along with localized memories, a standard set of peripherals and support for the RISC-V debug specification [12]. We evaluate the real-time performance of *Atalanta* based on post-implementation simulation results and compare the performance to the state-of-the-art.
- We implement prototypes of *Atalanta* on a low-end and high-end FPGA board, the Xilinx PYNQ-Z1 and VCU118, respectively. We present the relative area footprint of the design as a utilization percentage on the PYNQ-Z1 FPGA target.
- We implement preliminary software support for CLIC in Rust and publish it under a permissive open-source license⁵.

2 Background and Related Work

This section introduces the RTIC framework and presents the benefits of its adaption in system design. We provide an overview of the established interrupt handling schemes in RISC-V and define interrupt latency as the primary metric for performance in the context of this research.

2.1 Real-Time Interrupt-Driven Concurrency Framework

The RTIC framework is a domain-specific language (DSL) extension to the Rust language, providing a declarative, executable model for hard real-time applications. RTIC adopts the notion of concurrent tasks with shared resources in compliance to the SRP model for preemptive scheduling on single core processors. The RTIC framework is implemented as a Rust procedural macro, which performs model compliance checks, analysis and code generation. The framework seamlessly integrates into the Rust ecosystem with the aim to provide an effective and easy to use workflow. At run-time, RTIC applications (unless explicitly marked `unsafe`) are ensured a set of formal properties by the framework:

- Due to the Rust language:
 1. memory safety,
 2. absence of data races, and
 3. defined behavior.
- Due to the RTIC framework model and implementation:
 1. deadlock-free execution,

³ <https://github.com/soc-hub-fi/rt-ibex>

⁴ <https://github.com/soc-hub-fi/atalanta>

⁵ <https://github.com/rust-embedded/riscv/pull/195>

2. single shared stack execution among all tasks,
3. single dispatch (once started, a task will never yield), and
4. bounded priority⁶ inversion⁷.

Regarding real-time capabilities, RTIC is both less and more than expected from an operating system or kernel. It is less in that the set of tasks and resources are fixed at compile-time, i.e., there is no way to dynamically create new tasks or introduce new resources at run-time. It is also less, in that one cannot call into the kernel, i.e., there is no operating system at run-time – resource protection and scheduling primitives are fully inlined. At the same time, RTIC is more than an operating system. Properties and guarantees hold for the complete application, not merely the kernel. Notably, RTIC renders applications being correct by construction and free of misuse. Any attempt (due bug or malicious intent) to break with Rust or SRP invariants will either be caught at compile time or at run-time by Rust-injected run-time checks for cases out of reach of static analysis.

During code generation, tasks are mapped directly to interrupt handlers, allowing the scheduling to be performed directly by the underlying hardware. Thus, scheduling is zero-cost, and in fact, as fast as the interrupt handling mechanism of the hardware without *any* added overhead.

Resource protection boils down to a single `lock` primitive. The user-facing API is a function `lock` taking a closure argument:

```
cx.shared.r.lock(|r| /* user code with access to r */);
```

The lock causes the resource access to be guarded by way of raising the SRP system ceiling:

$$II = \max(II, \lceil r \rceil) \tag{1}$$

The implementation of the methods for setting the system ceiling and the maximum system ceiling is target dependent. For example, the ARM Cortex[®]-M3 and above provide an interrupt threshold register `BASEPRI` supporting `read`, `write` and `write_max` operations. This boils down to an effective lock/unlock of only three machine operations and a register pressure of one for storing the old value. For targets without threshold functionality, e.g., the Cortex[®]-M0/M0⁺, the `system_ceiling` can be implemented by pre-computed enable and disable masks.

The CLIC implementation adopted for this paper specifies `mintthresh` – a read/write control-and-status register (CSR) providing an atomic get/set. This allows CLIC based designs to complete the lock/unlock cycle in two machine

⁶ The word *priority* is as a synonym for *level* as it is used in RISC-V specifications

⁷ Bounded priority inversion: under SRP, a task t with priority $p(t)$ is at worst blocked by the maximum critical section any lower priority task t_l holds a resource r , where $\lceil r \rceil \geq p(t)$

operations. Further optimizations are possible. The Rust compiler tracks mutations in sequential code, which can be leveraged in order to further optimize locking.

For cases where the current system ceiling equals or exceeds the resource ceiling the happy path completely avoids touching the underlying hardware. This will happen in case of nested locks where the outer lock has a higher or equal resource ceiling than the inner lock. In effect the inner lock becomes “lock-free”, which can be seen as a sub-zero cost of the lock abstraction.

Another advantage of this approach is that we never need to read the old ceiling value, as it is tracked by the compiler. In effect, the cost of a lock/unlock is now for the worst-case only two machine instructions with one register pressure and brought to zero for best-case lock nesting. Each task is wrapped into a trampoline, which reads/stores the ceiling value on entry/exit. This can be further optimized by reading/storing ceiling on demand, i.e., for tasks not touching the system ceiling, the cost will be eliminated.

2.2 Interrupt Architectures in RISC-V

The standard core-local interrupt behavior of RISC-V is defined in Volume II of the RISC-V ISA specification [13]. The generic name for an interrupt controller based on this specification is the core-local interruptor (CLINT). The CLINT provides simple inter-processor interrupts, timer functionality and preemption based on privilege level, but is commonly considered to be an insufficient standard for advanced embedded applications. This has motivated the development of the core-local interrupt controller (CLIC) extensions to the RISC-V privileged architecture [10]. The CLIC extension can be implemented to replace or extend the CLINT interrupt scheme with a total of 4096 interrupts per hardware thread. In CLIC-mode, each interrupt is controlled by four memory mapped control registers that functionally replace the interrupt pending and interrupt enable bits in the CLINT control and status registers (CSRs), as well as provide control for interrupt privilege mode, trigger type and level.

The interrupt scheme for multiprocessor RISC-V systems was initially defined in the platform-level interrupt controller (PLIC) specification. Similar to the CLINT, the PLIC contains inherent limitations that have motivated the development of a new platform-level interrupt architecture: the advanced interrupt architecture (AIA) [14].

The real-time performance of the CLINT scheme was evaluated and compared against an ARM Cortex[®]-M3 series microcontroller by Balas and Benini in [7]. Despite optimizing their software framework, a notable performance gap to the Cortex[®]-M3 was observed. The publication cited the CLIC as the most promising proposed interrupt scheme to close the microarchitectural gap to ARM.

Balas et al. followed this work up in [9] and [8], where they presented the implementation and integration of the CLIC to a RISC-V system running FreeRTOS [15]. The authors presented a *fast interrupt extension* with which they were able

to achieve task context switch and interrupt latencies comparable to the ARM Cortex[®]-M4 while targeting FreeRTOS.

2.3 Interrupt Latency

Interrupt latency is defined as the delay from the time an interrupt signal is visible to the microprocessor, to the execution of the first instruction in the interrupt handler, i.e, the first instruction following interrupt context-saving instruction block (since the latter instructions are associated with the "caller" and are positioned inside the handler due to the interrupts' asynchronous nature). For more formality, we decompose interrupt latency into the components:

$$\text{InterruptLatency} = T_a + (1 - c)T_s, \quad (2)$$

where

1. T_a := **interrupt acknowledge delay** – the time from the interrupt signal is active and visible to the CPU to the fetch of the first instruction in the handler. It encapsulates pipeline/prefetch buffer flushes and vector table access delays. T_a is a hardware-dependent quantity.
2. T_s := **context-saving delay** – the delay incurred from executing the instructions required for context-saving the "caller-saved" registers to memory. Even though context-saving can be offloaded to hardware, the set of executed instructions and the corresponding caller-saved set of registers are mainly dictated by the application binary interface (ABI). T_s is mainly an ABI-dependent quantity.
3. c := **interleaving factor** – a dimensionless quantity that describes the amount of context-saving delay T_s that is hidden in the interrupt acknowledge delay T_a .

For instance, systems relying on software context-saving have an interleaving factor of $c = 0$, which is reasonable since such interleaving is not possible unless the IF and ID stages are decoupled. At the other end of the spectrum, we find systems with ABI-independent context-saving techniques such as register banking. In the middle points of the space, we find systems with hardware accelerated context-save, typically extra logic that executes a set of ABI-dependent instructions hardwired in the pipeline's datapath.

This terminology provides a unified ground for comparison, encapsulates both hardware and software factors contributing to interrupt latency, and thus exposes the sufficient design decision to minimize it.

3 Architecture

The open-source *Ibex*, used in projects such as OpenTitan [16], is adapted as a starting point for our design. The selection of the *Ibex* over other alternatives, mainly the *CV32E40P* [17], used in [9] and [8], is motivated by the significantly

smaller size of the *Ibex* compared to the *CV32E40P*, as well as the smaller design complexity, which allows for low-effort customization. *Ibex* occupies an approximate area of 15-30 kilo-gate equivalents (kGE) in smaller configurations, [11], while the *CV32E40P* requires an area of 50-90 kGE depending on the exact configuration. The *CV32E40P* is a 4-stage core compared to the 2-stage *Ibex* and is more compute-oriented than our requirements dictate.

The targeted configuration in this *Ibex* instance is `rv32emc`, without the optional writeback register enabled. The use of the embedded E extension and EABI has been shown to be favorable in real-time applications compared to the integer instruction set in [7] and [9] due to the smaller number of caller-saved registers. The standard *Ibex* hosts a fixed-size interrupt interface with inputs for three CLINT-specified interrupts and 16 custom interrupts. To accommodate integration to the CLIC, the interface and internals of the CPU need to be extended.

3.1 RT-Ibex

To meet the requirements of *Atalanta*, *RT-Ibex* is created as a fork of the mainline *Ibex*. A block diagram of the new *RT-Ibex* is presented in Figure 1. Extending the *Ibex* with support for the CLIC requires modifications to the CSRs, controller, instruction fetch and instruction decode logic.

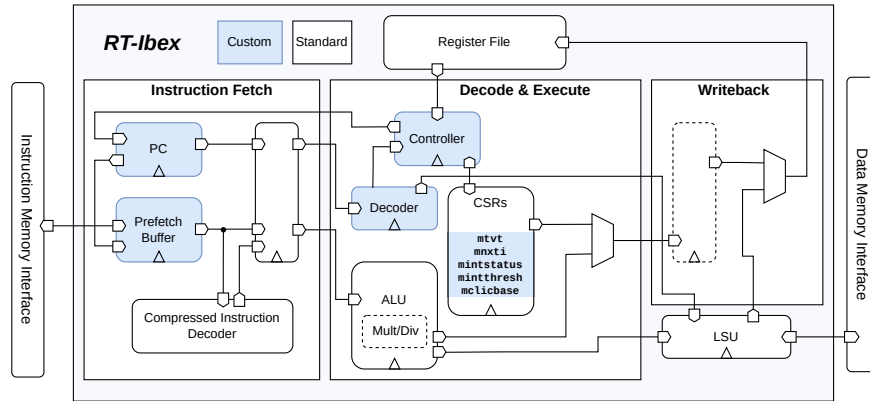


Fig. 1. Architecture of *RT-Ibex* (adapted from [11]).

Interrupt Interface The *RT-Ibex* completely replaces the fixed-size CLINT mode interrupt scheme of *Ibex* with a configurable number of CLIC interrupts. As the CLIC specification allows for 256 interrupt priorities per privilege level, a machine mode-only implementation of the core will still be able to assign unique priorities to all interrupt sources in smaller configurations.

Control and Status Registers To support the integration of CLIC, the specification defines an additional set of CSRs, as well as changes to existing CSRs. CLIC CSRs hold interrupt-handling related information for each hardware thread. Similar to the privileged ISA CSRs, the address space of CLIC CSRs is encoded with the 12 upper bits of CSR instructions, which enables atomic read, modify, and write. The CLIC specification defines 7 new CSRs per hardware thread for each privilege level. This implementation targets only machine-mode, hence the CSR unit is extended with the following registers:

- `mtvt`: base address of the Trap-handler vector table
- `mnxti`: interrupt handler address and enable modifier
- `mintstatus`: current interrupt level
- `mintthresh`: interrupt-level threshold
- `mclicbase`: base address for CLIC memory mapped registers (hardwired)

Furthermore, the `mcause` implementation is changed to comply with CLIC specifications, therefore `mcause` is extended to record more information about the interrupted context’s previous level and vectoring status, as well as aliased fields with `mstatus`.

Instruction Fetch Along with the controller, the instruction fetch (IF) stage features the most modifications. Our design objective was to enable CLIC interrupts with minor alterations to the *Ibex* instruction fetch logic by leveraging logic reuse.

In the *Ibex*, selection logic of the next fetch address (PC) requires two levels of multiplexing. The first level selects the appropriate type of `TRAP_PC`, interrupt or exception. The `TRAP_PC` is then fed to the second level that sets the fetch address to the trap address. For the first part of the problem, it’s sufficient to extend the first multiplexing level with the CLIC’s `vtable_entry_address` signal. The CLIC vector table entry address varies depending on the vectoring mode defined by the `clicintattr[i].shv` bit, where `i` is the interrupt number. Therefore, a third multiplexing level is necessary.

By default, the vector table entry is fetched as an instruction to the pipeline of the *RT-Ibex*. In the vectored mode that *Ibex* implements, the vector table entry holds a jump instruction to the trap-handler address. The resultant design is simple, however, it suffers some limitations. First, relying on RISC-V’s jump-and-link (JAL) instruction to trampoline to the handler restricts the address range to ± 1 MiB dictated by the 2’s complement signed 20-bit immediate bit-field. On the other hand, utilizing jump-and-link-register (JALR) instruction for full-address range necessitates more free registers, memory stack operations, and hence more instructions per vector table entry. Even if the ISA contains a full-range jump instruction or for the case where ± 1 Mib is enough for a given platform, embedding instructions in vector table negatively impacts interrupt latency since the microprocessor has to wait for a number of cycles equal to the number of its pipelines stages.

In our implementation, the vector table entries hold trap-handler addresses that are fetched as regular instructions. This raises several problems:

1. RISC-V is a low density ISA, meaning that the probability of the fetched trap-handler address causing an illegal instruction fault is high. Hence, the vector table entry should not enter the decode stage.
2. The *Ibex* supports the C compressed ISA extension with a compressed instruction decoder in the fetch stage. The compressed decoder either keeps the 4-byte instruction word unchanged, splits it into two compressed instructions, or issues an illegal instruction signal which is then handled by the decode stage with side effects on the fetch stage.
3. The fetched vector table is a pure 32-bit address value, containing no extra bit-field for the fetch stage logic to differentiate it from a regular instruction.

The proposed implementation solves the aforementioned problems as follows:

- An interrupt fires, the controller flushes the prefetch buffer, and a vector table entry is fetched and enters the prefetch buffer.
- The instruction decode stage is halted. Flushing the prefetch buffers does not zero-out the entries, rather, it zeroes-out the valid bits for all the entries. Since the buffer is flushed right before the vector table entry is fetched, the vector table entry can be identified as the first valid instruction after the flush, which solves problem 3.
- The vector table entry enters the compressed decoder. However, the decoder’s illegal instruction output signal is masked, solving problem 2.
- The fetched vector table entry is fed to the second multiplexer level, and PC is redirected to the trap-handler’s address on the next clock edge.
- The vector table entry is stored in IF/ID register and is thereby killed before reactivating the ID stage. This solves problem 1.
- Instruction decode stage is activated again and trap-handler code is executed.

The aforementioned steps are performed with pre-existing *Ibex* signals and logic with a minimal area overhead required for masking and multiplexing.

Instruction Decode Stage and Controller When an interrupt is visible, the FSM-based controller in the standard *Ibex* transitions to an interrupt handling state. However, the controller is forced to block waiting for a multi-cycle instruction in the ID-stage to complete, resulting in a jittered interrupt latency. The jitter value depends on the nature of the multi-cycle instruction. Multiply and divide instructions incur static worst-case latencies of 3 and 37, respectively. Other instructions, such as Jump and memory-mapped load/store, incur dynamic latencies starting from 1 clock cycle. As a design goal, T_a is minimized as much as possible while retaining zero-jitter. Note that utilizing low-latency memories for the entire memory address space does not guarantee single-cycle loads and stores, since the latter operations can be executed on other memory-mapped peripherals connected through a bus interconnect. In fact, the only multi-cycle instructions benefiting from zero-jitter low memory latency are jump instructions. However, jump instructions still execute in multi-cycle fashion when prefetch delays after the jump are added.

To resolve the jitter issue, the *RT-Ibex* controller aborts multi-cycle instructions to handle the interrupt in the next clock edge. In this case, functional units are first terminated to release the IF-stage before being idled with no changes being committed to the architectural state. Simultaneously, the controller saves the corresponding PC (PC ID) to the MEPC CSR. On the other hand, both the single-cycle instructions and multi-cycle instructions that finish in the next clock edge are allowed to commit results to the architectural state while saving PC IF to the MEPC CSR. Misaligned memory accesses are not single cycle even for low-latency memories, and therefore it may change the architectural state before aborting. However, this is tolerated since double-access to low-latency memory regions don't produce side effects that might hinder system's correctness. *Atalanta's* memory mapped peripheral registers are 4-bytes aligned and hence modified atomically. However, *RT-Ibex* does not distinguish strongly ordered memory accesses, which should be handled by the application.

3.2 Microcontroller

The architecture of the proposed *Atalanta* microcontroller is presented in Figure 2. The memory architecture of *Atalanta* is optimized for minimal latency memory access to the instruction memory (IMEM) and data memory (DMEM). The rationale of very low-latency memory operations is to significantly reduce the latency of context switches that require the CPU registers to be pushed and popped from the stack. The IMEM and DMEM of *Atalanta* are implemented in the memory subsystem with true dual-port (TDP) memory macros along with small bootROM. This allows for direct, low-latency access from the CPU, while also enabling backdoor access to the memories for initialization and debug.

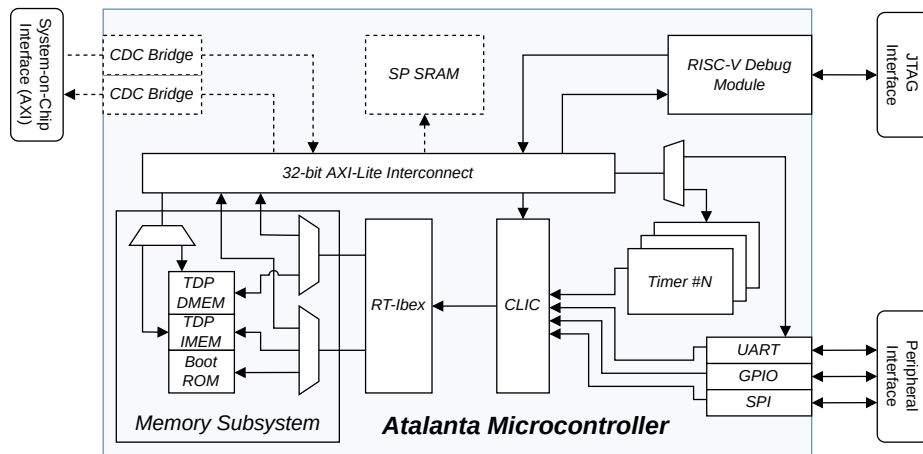


Fig. 2. Block diagram for the proposed *Atalanta* microcontroller.

The general connectivity of the system is implemented with an open-source AXI4 crossbar implementation [18], while the debug access is implemented with an open-source implementation of the RISC-V debug module [19]. The CLIC is based on the same open-source implementation as in [9] and is driven by a set of timers and peripherals. The AXI interconnect allows for the system to be easily extended, for example, with additional memories in the form of single-port (SP) SRAM for additional low-performance capacity. *Atalanta* is designed as a single clock domain system. Integration to a larger System-on-Chip (SoC) requires synchronization of clock domains, thus AXI clock domain crossing (CDC) bridges are supported in an optional configuration.

The architecture of *Atalanta* is notably very bare and does not currently implement any memory management or protection units. This is in line with the design philosophy and the requirements of RTIC, as the programming model can formally guarantee memory safety at compile-time [20]. Furthermore, we target a machine-mode-only system. This simplifies the hardware architecture and is enabled by RTIC, as the framework manages fine-grained, per-task access control at compile-time.

4 Evaluation

This section presents the FPGA implementation of *Atalanta*, an analysis of its real-time performance and a comparison to relevant commercial and academic state-of-the-art microcontrollers.

4.1 Implementation

Atalanta was synthesized and implemented in Vivado Design Suite 2021.2 with a frequency of 45 MHz for the Xilinx PYNQ-Z1 and the Xilinx VCU 118 FPGA to create prototypes for application development and to evaluate the implementation characteristics of the design.

Area The area overhead of the *RT-Ibex* compared against the original *Ibex* when using a flip-flop (FF) based register file (RF) is presented in Figure 3. The area requirement of the *RT-Ibex* is approximately 10 % more than the original *rv32emc Ibex*, with the extended CSRs contributing the majority of the difference. The area effect of the writeback register stage was found to be negligible in this comparison.

The area of the entire *Atalanta* microcontroller is mainly dependent on the configuration of memory sizes in number of interrupt inputs. The CLIC can, by specification, be scaled to a maximum of 4096 interrupt inputs with 8-bit priorities. This configuration is however impractical, especially for microcontroller-class devices. This is due to the required $4 * 8$ -bit registers per interrupt source, as well as the inferred arbitration logic. While the former scales linearly with the number of interrupt inputs, the arbitration can be implemented as a binary tree as in [9] to scale logarithmically. It is therefore desirable to explore configurations

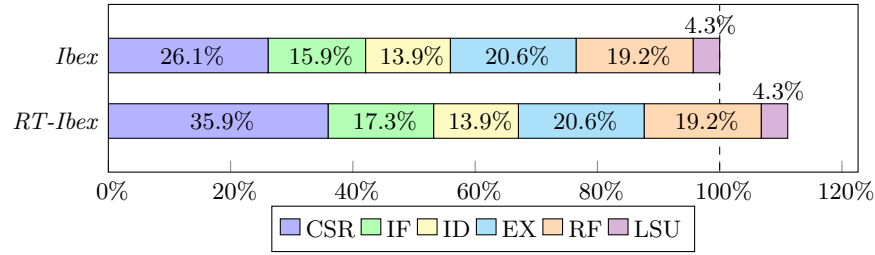


Fig. 3. Area breakdown of *RT-Ibex* relative to original *Ibex*.

of the CLIC with fewer interrupt inputs and interrupt levels. On the other hand, 32 interrupt inputs are the minimum feasible amount of interrupt inputs when the bottom 16 are reserved for CLINT comparability. This configuration would at face value be comparable to the original *Ibex*, albeit with fully configurable interrupt priorities.

The area requirement for *Atalanta* can be evaluated by considering its utilization when implemented on the Xilinx PYNQ-Z1, a small, low-end FPGA board. In a configuration with 64 interrupt sources and 8 kB of total memory, *Atalanta* utilizes 12395 lookup-tables (LUTs) (23.3 %), 6809 FFs (6.4 %) and two BRAM tiles (1.4 %).

Critical Path The bottleneck for the clock frequency on the FPGA implementation is within the CLIC, more precisely the combinational binary tree for interrupt arbitration. As the depth of the tree depends on the total number of inputs, the critical path is also determined by this configuration.

The implementation of the memory subsystem with BRAM macros allows for single cycle memory accesses without a detrimental effect on the timing characteristics of the system. The multiplexed paths from the *RT-Ibex* to the AXI crossbar require the crossbar to be configured to reduce combinational path length to meet the timing constraint of 45 MHz.

4.2 Analysis

The following measurements are based on RTL simulations and are validated with FPGA post-synthesis and post-implementation simulations, in addition to FPGA prototyping. The *Atalanta* microcontroller is simulated with a program loaded to memory. The main program consists of a simple infinite loop with multi-cycle instructions: load and store to address ranges outside the local memory, multiplication, and division with read-after-write dependence. The program enables and pends a CLIC interrupt. The interrupt threshold is lowered to disable interrupt-masking on the *RT-Ibex*, the point in time from which the measurements start.

Memory Latency Memory operations on *Atalanta* can access the memory subsystem directly or the rest of the memory map through the AXI interconnect.

The demultiplexing of the *RT-Ibex* memory interfaces requires stalling the memory request for one cycle if a switch from one memory region to the other occurs to conform to the load-store unit (LSU) protocol. The measurements show that AXI memory accesses have a maximal latency of 6 cycles, while local accesses have a latency of 0 or 1 cycles, depending on the state of the demultiplexer.

Instruction Abort For T_a , we measure a delay of 5 clock cycles. The multi-cycle division instruction is aborted one clock cycle after the interrupt is visible to the *RT-Ibex*, three of which required to fetch the vector table entry from memory and jumping to the Trap-handler code.

Context Switch Unlike ARM Cortex[®]-M-series, *RT-Ibex* relies on purely software context saving. The EABI defines 7 caller-saved registers for function calls. For interrupt handlers, `mcause` and `mret` are caller-saved since they store interrupt context information such as the return address and interrupt level. Using EABI interrupt handlers requires 12 instructions, 9 of which are memory stores to DMEM. We measure T_s to be 16 clock cycles.

4.3 Comparison

Table 1 presents the performance characteristics of *Atalanta* along with a selection of both commercial and academic state-of-the-art microcontrollers. The listed ARM cores accelerate the interrupt context saving with either hardware stacking or register banking. This reduces the instruction count and gives a total interrupt latency of 12 to 20 cycles, depending on the exact platform. RISC-V-based cores with software stacking achieve a latency of 20 to 24 cycles. Our design achieves equivalent performance to the software-based designs and approaches the performance range of the ARM Cortex[®]-series.

Table 1. Comparison of *Atalanta* with relevant commercial and academic state-of-the-art real-time-oriented microcontrollers.

MCU	ISA	Interrupt Controller	Interrupt Context-Save	ABI	instruction Count	T_a	T_s	c	Interrupt Latency
Cortex [®] -M0	ARM	NVIC	Hardware Stacking	AAPCS	9	6	n.a	n.a	16
Cortex [®] -M0 ⁺	ARM	NVIC	Hardware Stacking	AAPCS	9	6	n.a	n.a	15
Cortex [®] -M3/4	ARM	NVIC	Hardware Stacking	AAPCS	9	6	9	0.33	12
Cortex [®] -R5	ARM	VIC/GIC	Register Banking	AAPCS	9	n.a	n.a	n.a	20
SiFive E21	RISC-V	CLIC	Software Stacking	n.a	n.a	n.a	n.a	n.a	20
Balas et al. [7]	RISC-V	CLINT	Software Stacking	EABI	12	6 ^a	18	0	24
Balas et al. [9]	RISC-V	CLIC	HW Stacking + Register Banking	EABI	12	6 ^a	18	1	6 ^b
This Work	RISC-V	CLIC	Software Stacking	EABI	12	5	16	0	21

^a Susceptible to jitter from multi-cycle instructions prior to vector table entry fetch.

^b Interrupt latency can increase in the case of nested interrupts.

4.4 Discussion

As can be seen in Table 1, the practical performance limit of software-based context switching is around 20 clock cycles. Hardware stacking, register banking, or a combination of both has been shown effective in reducing the interrupt latency to below 20 cycles. Outside conventional methods, a potential solution to achieve interrupt latency with *zero-latency* is *register windowing*. Alluded to already in [21], redundant hardware registers dedicated for specific tasks could allow for context switching without the need to explicitly save and restore the content of the register file. The challenge of register windowing is guaranteeing the interrupt latency if and when the level of interrupt nesting exceeds the number of physically available register windows.

While low-latency performance is generally desirable for control-oriented applications, the evaluation of best-case or common-case latencies is inherently not valuable for WCET calculations in real-time systems. Future work on *Atalanta* should consider thorough worst-case timing analysis for applications on the systems. The implantation of *Atalanta* occupies a small area footprint on a low-end FPGA. However, this is a ballpark-estimate at best, and the ASIC implementation should be explored, both as a continued feasibility study and to produce more comparable results to the work presented in [9]. Additional targets of the continued development of *Atalanta* are conformance to the official RISC-V Architecture Test [22], the addition of experimental features to surpass the performance of the current state-of-the-art, improving the maturity of the design for ASIC implementation and demonstrating the performance of the system with prototype RTIC applications.

5 Conclusions

In this publication, we present the *RT-Ibex*, a fork of *Ibex* with support for interrupt-level-based preemption and a CLIC-compliant interrupt interface, and *Atalanta* – the first RISC-V microcontroller designed around the RTIC framework. We evaluate *Atalanta* by means of FPGA implementation, timing analysis and comparison to the state-of-the-art. We are able to demonstrate a hardware interrupt latency of 5 clock cycles and a total context switch latency of 21 clock cycles, which matches the best software-based context switch implementations and is near the performance of the ARM Cortex[®] series. We implement prototypes of *Atalanta* on the Xilinx PYNQ-Z1 and VCU118 FPGA boards, targeting 45 MHz. We make the hardware and software sources and implementation scripts of *RT-Ibex* and *Atalanta* available under a permissive open-source license.

Acknowledgments

This work was supported by the TRISTAN project, which has received funding from the Key Digital Technologies Joint Undertaking (KDT JU). KDT JU receives support from the European Union’s Horizon Europe research and innovation program.

References

1. Reghenzani, F., Massari, G., Fornaciari, W.: The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM computing surveys* **52**(1), 1–36 (2019)

2. Lee, E.: The Problem with Threads. *Computer* **39**(5), 33–42 (2006)
3. RTIC Contributors: RTIC: The Hardware-Accelerated Rust RTOS. <https://rtic.rs/2/book/en/> (2024)
4. Baker, T.P.: A Stack-Based Resource Allocation Policy for Real-Time Processes. *Proceedings 11th Real-Time Systems Symposium* pp. 191–200 (1990)
5. United States Cybersecurity and Infrastructure Security Agency: The Case for Memory Safe Roadmaps (2023), <https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf>
6. Waterman, A.: Design of the RISC-V Instruction Set Architecture. Ph.D. thesis, EECS Department, University of California, Berkeley (2016)
7. Balas, R., Benini, L.: RISC-V for Real-time MCUs - Software Optimization and Microarchitectural Gap Analysis. In: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 874–877 (2021)
8. Ottaviano, A., Balas, R., Bambini, G., del Vecchio, A., Ciani, M., Rossi, D., Benini, L., Bartolini, A.: ControlPULP: A RISC-V On-Chip Parallel Power Controller for Many-Core HPC Processors with FPGA-Based Hardware-In-The-Loop Power and Thermal Emulation. *arXiv.org* (2023)
9. Balas, R., Ottaviano, A., Benini, L.: CV32RT: Enabling Fast Interrupt and Context Switching for RISC-V Microcontrollers. (Preprint), [arXiv.org](https://arxiv.org) (2023)
10. RISC-V International: Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extensions. <https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc> (2024)
11. lowRISC: Ibex. <https://github.com/lowRISC/ibex> (2024)
12. RISC-V International: RISC-V External Debug Support. <https://riscv.org/wp-content/uploads/2019/03/riscv-debug-release.pdf> (2019)
13. Waterman, A., Asanovic, K., Hauser, J., RISC-V International: The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (2021)
14. Marques, F., Rodríguez, M., Sá, B., Pinto, S.: “Interrupting” the Status Quo: A First Glance at the RISC-V Advanced Interrupt Architecture (AIA). *IEEE Access* **12**, 9822–9833 (2024)
15. FreeRTOS: Market leading RTOS for embedded systems with Internet of Things extensions. <https://www.freertos.org/index.html> (2024)
16. lowRISC: Opentitan. <https://opentitan.org> (2024)
17. Schiavone, P., OpenHW Group: CORE-V CV32E40P User Manual. <https://cv32e40p.readthedocs.io/en/latest> (2024)
18. Kurth, A., Rönninger, W., Benz, T., Cavalcante, M., Schuiki, F., Zaruba, F., Benini, L.: An Open-Source Platform for High-Performance Non-Coherent On-Chip Communication. *IEEE Transactions on Computers* **71**(8), 1794–1809 (2022)
19. PULP-Platform: riscv-dbg. <https://github.com/pulp-platform/riscv-dbg> (2024)
20. Lindgren, P., Dzialo, P., Lunnikivi, H.: Hardware support for Static-Priority Stack Resource Policy based scheduling. In: 2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE). pp. 1–5. IEEE (2023)
21. Patterson, D.A., Séquin, C.H.: A VLSI RISC. *Computer* **15**, 8–21 (1982), <https://api.semanticscholar.org/CorpusID:5389873>
22. RISC-V International: RISC-V Architecture Test. <https://github.com/riscv-non-isa/riscv-arch-test> (2024)