CV32RT: Enabling Fast Interrupt and Context Switching for RISC-V Microcontrollers

Robert Balas, *Graduate Student Member, IEEE*, Alessandro Ottaviano, *Graduate Student Member, IEEE*, and Luca Benini, *Fellow, IEEE*

***Abstract*— Processors using the open RISC-V instruction set architecture (ISA) are finding increasing adoption in the embedded world. Many embedded use cases have real-time constraints and require flexible, predictable, and fast reactive handling of incoming events. However, RISC-V processors are still lagging in this area compared to more mature proprietary architectures, such as ARM Cortex-M and TriCore, which have been tuned for years. The default interrupt controller standardized by RISC-V, the core local interruptor (CLINT), lacks configurability in prioritization and preemption of interrupts. The RISC-V core local interrupt controller (CLIC) specification addresses this concern by enabling preemptible, low-latency vectored interrupts while also envisioning optional extensions to improve interrupt latency. In this work, we implement a CLIC for the CV32E40P, an industrially supported open-source 32-bit microcontroller unit (MCU)-class RISC-V core, and enhance it with `fastirq`: a custom extension that provides interrupt latency as low as six cycles. We call CV32RT our enhanced core. To the best of our knowledge, CV32RT is the first fully open-source RV32 core with competitive interrupt-handling features compared to the Arm Cortex-M series and TriCore. The proposed extensions are also demonstrated to improve task context switching in real-time operating systems (RTOSs).**

***Index Terms*— Context switching, embedded, interrupt latency, microcontroller unit (MCU), real-time, RISC-V.**

I. Introduction

SEVERAL markets, from automotive to aerospace and robotics, rely on real-time an SW-based solution [1]. For example, the automotive industry employs hundreds of electronic control units (ECUs) for real-time applications, such as electronic engine control, gearbox control, cruise control, anti-lock brake systems, and many other tasks.

General-purpose operating systems (GPOSs) are typically tuned for average throughput rather than real-time requirements imposed by such scenarios. For example, the development of Linux, a popular open-source GPOS kernel, is focused on average performance, making it less suitable to be used for real-time applications [2].

Albeit extensions and modifications that aim at improving determinism and latencies of critical operations in Linux have been proposed and implemented [2], [3], [4], [5], and they do not guarantee strict bounds on maximum latencies of operations and lack industry-grade maturity to be employed in hard real-time scenarios.

Real-time operating systems (RTOSs) kernels are special-purpose operating systems (OSs) designed to provide real-time guarantees, such as task scheduling according to a given expected completion deadline and deterministic latencies of various operations [6].

The RTOS' scheduler might add a significant overhead due to the combined effect of both the *context switches* required to handle the transition from a foreground to a background task and the amount of time elapsed from the source event that causes the preemption and the first instruction of the awakened task (known as *interrupt latency* [7]), thus increasing the worst case execution time (WCET) [2], [8], [9].

The cost of saving and restoring the task state during a context switch is a significant concern as it remains relatively high. For instance, the process state that needs to be saved on a context switch includes the program counter, register files (RF)s, status registers, address space mapping, etc. Therefore, a significant number of memory access operations need to be performed to *store* the state of the preempted task and *restore* the state of the new task to be executed [10]. Long context switch times reduce available *task utilization* and the minimum viable switching granularity.

Besides the task's context switching, a switch into an interrupt context from normal program execution happens each time an asynchronous event is triggered from, e.g., an I/O peripheral device. Behnke et al. [11] provide a meaningful example of the impact on *interrupt latency* of network loads in time-critical microcontroller units (MCUs), where a high overhead generated by the receiving of packets can be seen in continuous floods as well as short transmission bursts, reaching up to 50% task *lateness*—the additional time a task takes to finish than its deadline allows, which will start to accumulate over iterations from the critical network load—increase per packet per second. Furthermore, HW-induced interrupt latency is only one part of the problem: SW-induced interrupt latency,

Manuscript received 18 August 2023; revised 25 January 2024; accepted 4 March 2024. Date of publication 21 March 2024; date of current version 23 May 2024. This work was supported in part by the HORIZON Key Digital Technologies Joint Undertaking (KDT JU) Programme through the TRISTAN project under Grant 101095947. *(Corresponding author: Robert Balas.)*

Robert Balas and Alessandro Ottaviano are with the Integrated Systems Laboratory (IIS), ETH Zürich, 8092 Zürich, Switzerland (e-mail: balasr@iis.ee.ethz.ch; aottaviano@iis.ee.ethz.ch).

Luca Benini is with the Integrated Systems Laboratory (IIS), ETH Zürich, Zürich, Switzerland, and also with the Department of Electrical, Electronic and Information Engineering (DEI), University of Bologna, 40126 Bologna, Italy (e-mail: lbenini@iis.ee.ethz.ch).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2024.3377130.

Digital Object Identifier 10.1109/TVLSI.2024.3377130

introduced by the GPOS/RTOS scheduler and the user code, primarily impacts the capability of the system to provide timely responses to asynchronous events.

Low interrupt latency and context switch time are crucial metrics for a wide range of platforms ranging from commodity MCU-class embedded systems to more advanced and complex application-class mixed criticality systems (MCSs), where time/safety-critical and non-critical applications coexist on different isolated partitions of the same HW platform [12].

Response and *context switch* time minimization thus become a challenge to be tackled at the HW/SW interface, where SW programming techniques and HW *interrupt controller* architectures can cooperate to ensure minimal response time [7].

Although commercial vendors and IP providers offer such features as in-house solutions, they are often proprietary and tightly coupled with the vendor's instruction set architecture (ISA), target HW family, and associated SW stack. On the other hand, in the last decade, the ever-growing RISC-V ecosystem [13] has been offering a modular, free, and open-source ISA which is rapidly becoming the *de facto lingua franca* of computing.

Nevertheless, RISC-V support for fast interrupt and context switch handling is still not mature enough to compete with incumbent proprietary architectures, as it lacks flexible interrupt prioritization, preemption mechanisms, and low interrupt latency. For this reason, the RISC-V community has been developing an extension to the *Privileged* specifications [14] with the proposal of the RISC-V core local interrupt controller (CLIC) [15], currently under ratification by the community, to handle such real-time scenarios.

While the modular RISC-V ISA enables developing orthogonal *custom extensions*, to the best of the authors' knowledge, there are few published works that 1) tackle the problem of minimizing *interrupt latency* and *context switch* time from a holistic (HW and SW) viewpoint for RISC-V; 2) try to close the gap with more established proprietary solutions, thereby promoting RISC-V as a valuable candidate for time- and safety-critical application domains such as automotive and aerospace; and 3) provide an open-source solution to be shared with the community.

To address these open challenges, we propose CV32RT, a 32-bit RISC-V core that extends the interrupt handling capabilities of CV32E40P [16], [17], an industrially supported open-source core, to achieve best-in-class interrupt latency and fast context switching against commercial off-the-shelf (COTS) processor vendors, paving the road for RISC-V architectures in time-critical systems.

In particular, this article makes the following contributions.

1) We replace the CV32E40P interrupt controller with an implementation of the RISC-V CLIC specification, which provides the core with key features such as prioritization by level and priority, selective hardware vectoring (SHV), and non-nested interrupt optimization [*tail-chaining* through the `xnxti` [15] control and status register (CSR)] directly as RISC-V standard extension (see Section III-A). We make the implementation available under a permissive open-source license.[1]
2) We design a *fast interrupt extension* (`fastirq`) to accelerate both nested (see Section III-A) and non-nested (see Section III-D) interrupt case scenarios. `fastirq` reduces *interrupt latency* by hiding the latency through memory banks and a background-saving mechanism. The same mechanism allows one also to accelerate *context switching* through HW/SW cooperation. Furthermore, we propose `early mret` (`emret`), a novel instruction that further optimizes tail-chaining compared to the baseline strategy proposed in the CLIC standard (i.e., `xnxti`) and its enhancement from [18] (`jalxnxti`).
3) We integrate CV32RT within an open-source system on chip (SoC) [19].[2] We evaluate CV32RT interrupt handling capabilities while showing the negligible area overhead introduced by the extension in a modern technology node against its performance benefits (see Section IV).
4) Finally, we compare CV32RT with leading COTS systems in both nested and non-nested interrupt scenarios (see Section V). We show that the proposed solution promotes RISC-V as a competitive candidate for building the next generation of time-critical systems.

II. Background

We give a brief overview of the full-system platform used to design and implement the proposed interrupt extension in Section II-A, motivate and explain the relevant target metrics in Section II-B3, and describe the current status of interrupt handling in RISC-V in Section II-C.

A. RISC-V System Platform

We rely on the CV32E40P core [16]—abbreviated to CV32 in this article—an open-source, industry-grade, 32-bit, in-order, four-stage RISC-V core, as the basis for implementing our extensions. This core is embedded in ControlPULP [19], a SoC specialized in running real-time workloads. The system contains a CV32 manager core, a programmable accelerator subsystem consisting of eight CV32 cores, and a set of standard peripherals such as quad serial peripheral interface (QSPI), inter-integrated circuit (I2C), and universal asynchronous receiver-transmitter (UART). The manager core is responsible for scheduling tasks, communicating with the peripherals, offloading tasks to the accelerator subsystem, and being responsive to asynchronous external events, e.g., interrupts.

ControlPULP hosts a set of scratchpad memories (SPMs) that guarantee *single-cycle access time* from the CV32 manager core. This design choice enables deterministic memory access latency for both data load, store, and instruction fetch, bounding the worst case latency when handling unpredictable events.

Applications run on top of FreeRTOS [20], an open-source, priority-based preemptive RTOS, in the manager core from where tasks are scheduled and run.

[1] https://github.com/pulp-platform/clic

[2] https://github.com/pulp-platform/control-pulp

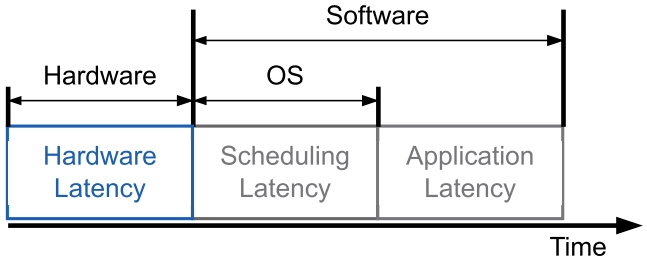


Fig. 1. Interrupt latency breakdown into SW- and HW-dependent contributions.

B. Interrupts

1) Level- and Edge-Triggered Interrupts: Interrupt sources can signal interrupts either through a level change of the interrupt line, called a *edge-triggered interrupt*, or by the logic level itself, a *level-triggered interrupt*. While the latter requires the receiving side of the interrupt to clear the source often through accessing appropriate HW registers, the former is unidirectional notification without confirmation. This results in faster processing but may lead to dropping interrupt requests accidentally.

2) Vectored and Direct Interrupts: Interrupts are asynchronous events that alter the *normal program order execution* and require a switch to a different context to handle the event. A processor supports *vectored* interrupts when each interrupt traps to a specific interrupt service routine (ISR) according to an *interrupt vector table*, granting fast interrupt response at the cost of increased code size. Conversely, *non-vectored* or *direct* interrupts trap to a shared ISR. The latter approach trades code size off for a slower interrupt response since the overhead of resolving the interruption cause and jumping to the correct ISR are handled in explicit instructions while the interrupt table can be made much more compact.

3) Interrupt Latency: Multiple sources determine the interrupt latency in a system: the underlying HW, the scheduler or OS, and the application running on top as detailed. This breakdown is detailed in Fig. 1.

In this article, we focus on minimizing the latency imposed by the HW; thus, whenever we refer to interrupt latency, we mean its HW-contributed part. Interrupt latency is defined as the time it takes from an interrupt edge arriving at the HW, usually the interrupt controller, to the execution of the first instruction of the corresponding interrupt handler routine. To make a fair comparison between SW-based and more HW-oriented interrupt solutions, we have to delineate what exactly constitutes the first instruction of the interrupt handler. We count as the first instruction the one after all necessary interrupt context has been saved on the stack to be able to call a *function*. What exactly entails a function call is dictated by the used application binary interface (ABI) or, more precisely, its calling convention. Note that when the interrupt handler and interrupt context saving code can be interleaved, some of the context saving code can be removed as it might be redundant.

4) Context Switching Time: The time it takes for a context switch determines the responsiveness of the architecture in swapping from one execution context to another. The execution context is dependent on the OS being used but usually consists of the *state* of the architectural registers of the ISA and the chosen ABI.

5) Interrupt Nesting: Preemption refers to an event, such as an interrupt request, temporarily interrupting a current task with the purpose of resuming its execution later.

The simplest case for preemption occurs with *non-nested interrupt handlers*, where interrupts are globally disabled during the execution of an ISR. In the case of level/priority interrupt schemes, this means that: 1) if the level/priority arbitration is SW-driven, e.g., with priority simple/standard interrupt handlers, the highest priority interrupt identification code is not executed, and 2) if the level/priority arbitration logic is designed within the interrupt controller, the highest level/priority interrupt is pending but disabled, hence not propagated to the core. This scenario is not ideal for real-time and complex embedded systems, as interrupts are served sequentially. A high-priority interrupt has to wait for a lower-priority interrupt to finish.

A more complex case for preemption occurs with *nested interrupt handlers* to handle the case of multiple interrupts at a time. In this scenario, interrupts are globally enabled within the scope of an executing ISR. The ISR need to care designed to ensure the they are reentrant. With a level/priority interrupt scheme (either SW or HW driven), this introduces additional masking of incoming interrupts of equal or lower level/priority than the executing ISR, and sometimes larger than a configurable level/priority threshold. The nesting allows higher priority interrupts to preempt a current lower priority ISR executing.

6) Redundant Interrupt Context: Back-to-back interrupts, i.e., interrupts that need to be served sequentially, can happen whenever there are multiple interrupts pending. The transition from one interrupt to the next one causes a redundant sequence of context restores and context saves. The active interrupt handler's context must be restored on interrupt return but is immediately saved again due to the next pending interrupt firing. These redundant context restore sequences negatively impact interrupt latency on higher interrupt loads [21]. Section III-D explores the optimizations implemented in this work to address this scenario.

C. Interrupt Handling in RISC-V

1) CLINT and PLIC: We distinguish between CLICs, local to each hardware thread (HART), and *platform level interrupt controllers* (PLICs), centralized interrupt controllers capable of managing multiple HARTs. In the RISC-V ecosystem, the privileged specification [14] defines a simple interrupt scheme with a set of timer and inter-processor interrupts. We refer to it as the core local interruptor (CLINT). For 32-bit cores, it defines a fixed priority interrupt scheme with 16 predefined or reserved and 16 implementation-defined interrupts that can be optionally vectored. The CLINT itself only supports prioritization of interrupts based on privilege mode. This is insufficient for the embedded and real-time domain, and attempts to handle a more complex interrupt scheme would require SW emulation, which incurs untenable interrupt latencies. To increase the number of custom interrupts, a PLIC [22] can be attached to the CLINT. It can route a

TABLE I
NESTED INTERRUPT PREEMPTION SCHEME ACCORDING TO RISC-V CLIC. WE ASSUME AN INTERRUPT REQUEST `IRQ2` IS BEING SERVICED WHILE ANOTHER REQUEST `IRQ1` IS ENABLED AND PENDING. PREEMPTION IS REGULATED BY PRIVILEGE MODE (*Vertical* INTERRUPTS) AND INTERRUPT LEVEL WHEN THE PRIVILEGE MODE IS THE SAME (*Horizontal* INTERRUPTS)

Nested interrupt behaviour	Condition	CLINT preemption?	CLIC preemption?
Vertical	$\texttt{priv1} > \texttt{priv2}$	✓	✓
Horizontal	$(\texttt{priv1} == \texttt{priv2}) \wedge ((\texttt{l1} > \texttt{thresh}) \wedge (\texttt{l1} > \texttt{l2}))$	✗	✓

flexible (at design time) number of interrupts to one or more targets. A single HART can correspond to multiple targets. Each interrupt can be assigned a priority, and each target can select a threshold below which interrupts are disabled. This scheme allows interrupts to be divided according to their priorities on the PLIC-level allowing for some flexibility in terms of prioritization, but does not address the flexibility problem on the core local-level.

2) CLIC: The CLIC [15] addresses these limitations by allowing interrupts to be prioritized by so-called *levels* and *priorities*. Interrupt selection is driven by the CLIC in HW (see Section III-B), which propagates the highest level, highest priority pending interrupt to the core's interface. Compared to standard RISC-V privileged specifications, pending and enabled interrupts are further selectively masked according to a *threshold value* representing an interrupt level, configured through a CSR. Interrupts that are enabled, pending, and have a level below the threshold are masked, while others can be propagated. This feature is useful for, e.g., RTOSs that only want to disable a subset of all interrupts during critical sections. Interrupts that do not interfere with the data accessed in such a critical section can still fire.

Consider two interrupts' requests `irq1` and `irq2` with interrupt privilege modes, levels and priorities (`priv1`, `l1`, `p1`) and (`priv2`, `l2`, `p2`), respectively. Following the terminology introduced in Section II-B and by the CLIC, we will refer to interrupts fired from different privilege modes as *vertical* interrupts and to interrupts fired from the same privilege mode as *horizontal* interrupts throughout this article.

Table I shows preemption conditions of two nested interrupts `irq2` and `irq1` according to the CLIC specification. We consider an active interrupt handler servicing `irq2` while `irq1` is pending. In general, `irq1` preempts `irq2` whenever its level is higher than both the interrupt threshold and `irq2`'s level. Interrupt priority serves as a tie-breaker for the case of multiple interrupts `pending` with the same level. This situation does not result in preemption, rather causes pending interrupts to be serviced in sequence according to increasing priority. Analogously, the case where multiple horizontal interrupts have equal levels and priorities results in the CLIC selecting the highest numbered interrupt (identification number `id`) [15], which is an arbitrary assignment decided at design time.

Finally, the CLIC specification addresses the case of *redundant context restore* (see Section II-B6) by introducing `xnxti`, a CSR short for *Next Interrupt Handler Address and Interrupt-Enable CSRs* meant for use with non-vectored interrupts. Reading from this CSR, while the core is within an active handler, allows to fast-track interrupts that arrive late or to avoid redundant context save/restore by running through pending interrupts back-to-back.

III. ARCHITECTURE

Entering an interrupt context or performing a context switch requires the HW to store enough information to resume operation correctly after returning from the aforementioned context. This needs to be done as fast as possible. From a high-level point of view, we can effectively improve interrupt latency and context switch times by:

1) controlling the amount of state that needs to be preserved to enter and leave an interrupt context;
2) increasing the bandwidth and decreasing the latency to memory;
3) relying on *latency-hiding* techniques that defer the effective saving of the state to a later point in time.

In the following, we detail the architectural features and HW/SW codesign of CV32RT (see Sections III-A–III-C), and then describe how the proposed architecture tackles typical case scenarios (see Section III-D).

A. CV32RT Overview

We start with the CV32 core as the baseline, whose native CLINT interrupt controller we replace with the CLIC (hereafter, CV32RT$^{\texttt{CLIC}}$), and then introduce `fastirq` into the core microarchitecture, the extension proposed in this work (hereafter, CV32RT$^{\texttt{fastirq}}$). CV32RT$^{\texttt{fastirq}}$ optimizes interrupt latency in the non-nested interrupt case by combining bank switching and the nested interrupt case with an automatic context-saving mechanism in the background. The background-saving mechanism updates the stack pointer and stores the bank-switched contents in memory while the execution of the core proceeds in parallel.

We describe each of these changes in detail below.

B. CV32RT^{CLIC}: CLIC Fast Interrupt Controller

We implement the CLIC interrupt controller in CV32RT$^{\texttt{CLIC}}$ [15] according to the draft specification which is to be included in the RISC-V Privileged Specification [14]. Fig. 2 provides an overview of the design. Incoming interrupts are filtered with a `Gateway` module that decides whether there is a pending and enabled request for each interrupt source *i* (`IRQ i`).

Assuming *n* interrupt sources, selecting the interrupt with maximum level and priority is implemented with three binary trees. For each interrupt source, they track: 1) interrupt level and priority; 2) interrupt `id`; and 3) pending state from the gateway module. The `Interrupt Prioritization` module traverses the tree from leaves to the root, where the sought-after maximum level and priority interrupt is found. Each tree has low overhead in terms of area and delay, $\mathcal{O}(n)$ and $\mathcal{O}(\log(n))$, respectively. The CLIC design proposed in

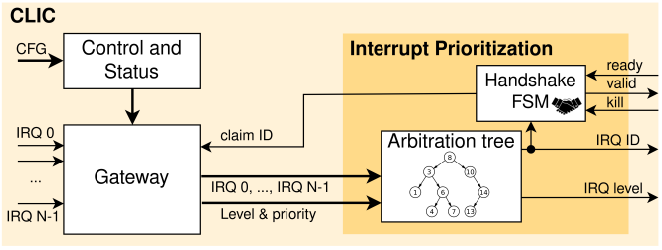


Fig. 2. CLIC architecture overview. Interrupts arrive at the gateway, where they are combined with programmable configuration information about each interrupt line consisting of level, priority, enable status, and sensitivity (level/edge). Enabled interrupts and their level and priority information are then further sent to prioritization logic which uses a binary arbitration tree to select the highest-level interrupt. The interrupt is then presented to the core with a handshake-based interface. The additional kill signal is there to allow for a handshake to restart so that a potentially more important interrupt can be presented to the core.

this work can scale up to $n = 4096$ local interrupt sources. Optionally, additional pipeline stages can be inserted in the arbitration tree to relax timing. Finally, our version of the CLIC supports SHV and the `xnxti` CSR in the core.

Albeit improving the interrupt handling capabilities of CV32 by introducing priority and levels management in HW, critical operations such as interrupt state and context save/restore are not natively covered by the CLIC and need to be handled in SW. CV32$^{\texttt{fastirq}}$ aims at filling this gap.

C. CV32RT$^{\texttt{fastirq}}$: Fast Interrupt Extension

1) Automatic Hardware Context Saving and Bank Switching: A block diagram of CV32RT$^{\texttt{fastirq}}$ is shown in Fig. 3. Conceptually, we can think of `fastirq` as a wrapper around the core's RF. We extend the RF by an additional read port for the background-saving mechanism and registers for latching the additional processor state required for proper interrupt nesting.

A new interrupt at the CLIC will first be checked whether the interrupt level exceeds the configured threshold. If this is the case, it will be dispatched to the core's main state machine while concurrently triggering the saving logic finite state machine (FSM) in the extended RF. The core's state machine will flush the pipeline and update the program counter according to the vector table entry. Meanwhile, in the extended RF, the saving logic triggers a bank switch, allowing the interrupt context to have a fresh set of registers while draining the other bank contents through a separate port to the main memory.

2) Stack Pointer Handling: During a bank switch, we need to update the stack pointer. For that, we have a dedicated adder between the two RFs. The RISC-V embedded and integer ABI dictate that the stack pointer points below the last saved register on the stack. Furthermore, if we did not adjust the pointer, the program code running in the interrupt handler could clobber the values on the stack. We could do away with this mechanism for leaf-type interrupts, but this would require a new ABI considering a virtual stack pointer offset when generating code for interrupt handlers, and it would not solve the nested interrupt case.

3) Conflict Handling: There are interactions between the background-saving mechanism and regular load/store instructions of the core that could potentially result in incorrect execution.

In the case where a load instruction is trying to update an architectural register while the background-saving mechanism is trying to read the same register, no conflict arises because interrupts are injected into the pipeline and only acted upon in the write-back stage of the core. At this point, updates to the RF are resolved, and then the bank switching operation takes place ensuring correctness of the execution.

While the interrupt handler is already executing, we also need to ensure that loads or stores accessing stack memory regions where the background-saving mechanism is writing to are properly resolved. Otherwise, we could read stale data or write data that is immediately overwritten. This is handled by informing the load-store unit of the core of the ongoing state being written to memory.

A straightforward solution is to stall the core's pipeline while the background-saving mechanism is at work. This, though, partially negates the advantage of executing ahead since we likely want to issue a load soon in the interrupt handler. We address this problem by additionally checking whether the load-store unit tries to access the memory in the range of the stack pointer. More precisely, on an interrupt the stack pointer is decremented, making space available, while the background-saving mechanism will start storing the interrupt state word by word. In the load-store unit, the address offset of the last word pushed out by the background-saving mechanism is compared against any incoming load and stores. Load and stores that try to access data that is not yet pushed to memory cause the core's pipeline to stall. This mechanism ensures the correctness of loads and stores issued by the core.

4) Co-Operation of Compiler and Hardware: Accessing stack memory locations during the execution of an interrupt handler typically happens in two use cases.

1) A system call handler (issued through the `ecall` instruction in RISC-V) wants to access user-provided arguments. Most will be passed through general-purpose registers, but some might be placed on the stack
2) Short interrupt handlers that want to return before the full interrupt state has been saved.

Each of these cases would potentially engage the stalling logic outlined in Section III-C3, causing higher interrupt latencies. An HW solution to mitigate the stalling issue would be to add forwarding logic that checks in the load-store unit what is being written by the background-saving mechanism and directly forward these values from the store queue instead of going through the memory subsystem outside the core. Some register values still cannot be forwarded because they might not have reached the load-store unit yet. Furthermore, the approach increases HW complexity, requiring a dynamic address lookup into a queue-like buffer.

We propose instead an SW-based solution that does not cause any kind of stalling. The insight is that we do not need to engage the pipeline stalling logic by ordering the loads to access the already stored interrupt state first. This can be achieved since we know how the interrupt state gets pushed

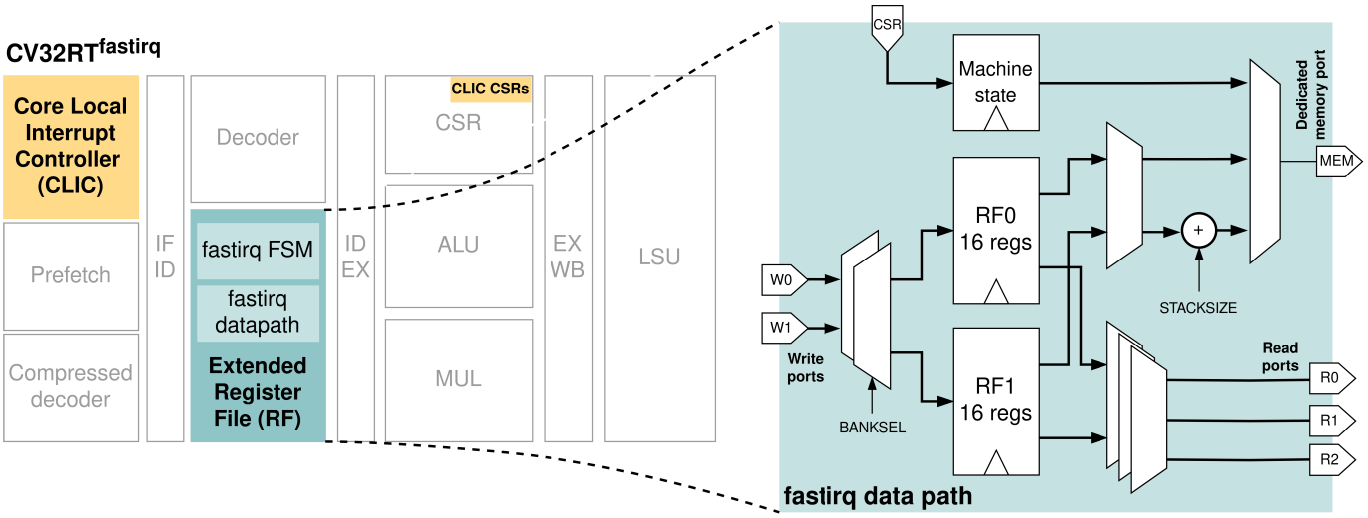


Fig. 3. Overview of the fast interrupt architecture. The cv32's RF is extended with additional logic for the background-saving mechanism. On an interrupt, the RF banks are switched. Parts of the old memory bank (the interrupt context) are copied to the core's stack location, while execution can go ahead by using the new bank. In this design, there is a dedicated memory port for the background-saving mechanism, but optionally it can also be shared with the port from the load-store unit. The stack pointer is adjusted appropriately to maintain ABI invariants.

to memory. For example, if we push out the general-purpose registers `x1`, `x2`, ... in that very same order, then we need to ensure that in the interrupt handler, we use the same order to load words back.

Note that this is a performance issue and does not affect correctness. When writing assembly, the programmer has to be aware of that to achieve the best possible latency. In the case that the programmer uses compiler-specific attributes to write his interrupt handlers, such as `__attribute__((interrupt))` in GCC, the compiler needs to be made aware of that fact.

5) Interrupt Handler Routines: In Fig. 4, we give an overview of how nested interrupt handling code works for the basic CLINT-mode, the baseline CLIC, and our `fastirq` extension. Due to the inflexible interrupt scheme of the CLINT-mode, much more work needs to be done in managing interrupt mask (`SOME_IRQ_MASK`) and other machine state. The CLIC addresses this with the level/priority scheme and `fastirq` improves upon that by moving the interrupt state saving logic in HW and adding `emret` to handle redundant interrupt context sequences.

6) Register File Overhead: RISC-V defines RV32E as a variant of RV32I, which only has 16 registers instead of 32. Reducing the RF size helps lower context switch times but does not affect the interrupt latency since the set of caller-save registers remains the same when using the embedded-application binary interface (EABI). Our implementation allows the core to dynamically switch between RV32I and RV32E with `fastirq` depending on the workload. This allows for the reuse of the unused half of the RF. If the increased pressure on the RF is not acceptable, instead of doubling the RF size for the banking logic, one could just add additional registers for the seven caller-save registers to save area.

D. Interrupt Scenario Analysis

1) Non-Nested Interrupts: As discussed in Section II-B, redundant context restore with non-nested or nested horizontal interrupts can introduce unwanted additional interrupt latency. With the proposed approach, the interrupt state can be quickly restored by simply switching register banks. To differentiate between a regular return from an interrupt handler using `mret`, which assumes the interrupt state has been restored by SW, we add a new instruction `emret` instead. This instruction performs the same function as `mret` in addition to switching register banks.

In the RISC-V ecosystem, this situation can be optimized by directly checking for other interrupts pending on the same level before restoring the execution's interrupt context. The CLIC offers an HW-assisted solution to address such a scenario with the `xnxti` CSRs (see Section II-C2). Nuclei's enhanced CLIC (ECLIC) [18] extends `xnxti` by embedding the jump to the queuing interrupt handler in the `xnxti` HW (`jalxnxti`). In our case, this is implemented through the `emret` instruction. Besides checking whether a bank switch is appropriate, it also looks at interrupts pending with the same or lower level. If such an interrupt exists, `emret` redirects the control flow to the pending interrupts handler. Some HW refers to this concept of removing *redundant context restores* as tail-chaining [23]. This situation is visualized in Fig. 5(b).

2) Nested Interrupts: Following the notation adopted in Section II-B5, the case of nested interrupts entails preemption of a low-level interrupt by a high-level interrupt, as shown in Fig. 5(a). We address this scenario by minimizing *interrupt latency* through hiding latency.

Whenever an interrupt handler is entered, global interrupts are disabled. This increases the worst case interrupt latency by the time they remain disabled. For nesting interrupts, the first instruction will be re-enabling global interrupts. At this point, a high-level interrupt could preempt the current running interrupt handler. If this happens while the background-saving mechanism is still at work, the execution of the handler has to wait.

This combination of latency hiding and background saving allows the core to quickly enter a first-level interrupt handler

CLINT Mode

```
save interrupt state
  addi  sp, sp, -12 * 4
  sw    a0, 1 * 4(sp)
  sw    a1, 2 * 4(sp)
  sw    a2, 3 * 4(sp)
  sw    a3, 4 * 4(sp)
  sw    t0, 5 * 4(sp)
  sw    t1, 6 * 4(sp)
  sw    ra, 7 * 4(sp)
  csrr  a0, mcause
  csrr  a1, mepc
  csrr  a2, mie
  sw    a0, 8 * 4(sp)
  sw    a1, 9 * 4(sp)
  sw    a2, 10 * 4(sp)
masking
  # compute SOME_IRQ_MASK
  # [...]
  csrw  mie, SOME_IRQ_MASK
  # re-enable global irq
  csrsi mstatus, 8
handler
  # interrupt handler
  # [...]
restore pre-interrupt state
  lw    a3, 4 * 4(sp)
  lw    t0, 5 * 4(sp)
  lw    t1, 6 * 4(sp)
  lw    ra, 7 * 4(sp)
  lw    a0, 8 * 4(sp)
  lw    a1, 9 * 4(sp)
  lw    a2, 10 * 4(sp)
  # disable global irq
  csrsc mstatus, 8
  csrw  mcause, a0
  csrw  mepc, a1
  csrw  mie, a2
  lw    a0, 1 * 4(sp)
  lw    a1, 2 * 4(sp)
  lw    a2, 3 * 4(sp)
  addi  sp, sp, 12 * 4
  mret
```

(a)

CLIC Mode

```
save interrupt state
  addi  sp, sp, -16 * 4
  sw    a0, 1 * 4(sp)
  sw    a1, 2 * 4(sp)
  sw    a2, 3 * 4(sp)
  sw    a3, 4 * 4(sp)
  sw    t0, 5 * 4(sp)
  sw    t1, 6 * 4(sp)
  sw    ra, 7 * 4(sp)
  csrr  a0, mcause
  csrr  a1, mepc
  sw    a0, 8 * 4(sp)
  sw    a1, 9 * 4(sp)
  # re-enable global irq
  csrsi mstatus, 8
handler
  # interrupt handler
  # [...]
restore pre-interrupt state
  lw    a3, 4 * 4(sp)
  lw    t0, 5 * 4(sp)
  lw    t1, 6 * 4(sp)
  lw    ra, 7 * 4(sp)
  lw    a0, 8 * 4(sp)
  lw    a1, 9 * 4(sp)
  # disable global irq
  csrsc mstatus, 8
  csrw  mcause, a0
  csrw  mepc, a1
  lw    a0, 1 * 4(sp)
  lw    a1, 2 * 4(sp)
  addi  sp, sp, 16 * 4
  mret
```

(b)

CLIC fastirq

```
  # re-enable global irq
  csrsi mstatus, 8
handler
  # interrupt handler
  # [...]
  # early return from irq
  emret
restore pre-interrupt state
  lw    ra, 1 * 4(sp)
  lw    t0, 2 * 4(sp)
  lw    a0, 3 * 4(sp)
  lw    a1, 4 * 4(sp)
  lw    a2, 5 * 4(sp)
  # disable global irq
  csrsc mstatus, 8
  lw    a3, 8 * 4(sp)
  lw    t1, 9 * 4(sp)
  csrw  mepc, a3
  csrw  mcause, t1
  lw    a3, 6 * 4(sp)
  lw    t1, 7 * 4(sp)
  addi  sp, sp, 9 * 4
  mret
```

(c)

Fig. 4. Routines for saving state for vectored nesting interrupts using (a) CLINT, (b) CLIC, and (c) proposed `fastirq` extension. This assumes the regular RISC-V embedded ABI. Note that in CLINT-mode interrupts with lower priority than the current interrupt running can fire when global interrupts are re-enabled. In order to prevent that, `mie` has to be manually adjusted such that the corresponding lower-priority interrupts are disabled.

and potential nesting interrupt handlers, albeit with a larger delay.

Restoring the pre-interrupt context is entirely handled in SW for the nested interrupt case. While this return path could also be handled in HW by adding an additional write port to the core's RF, exiting an interrupt handler is less time-critical since we mostly care about how quickly an external event is addressed.

E. Context Switching Acceleration

We can divide context switches into OS-specific and HW-specific parts. The OS part entails all contributions to the context switch time that is specific to the OS itself, such as computing the next task to be scheduled and bookkeeping operations. The remainder is the HW-dependent saving and restoring of the state belonging to the new context.

The idea is to use the background-saving mechanism to accelerate the state saving and restoring part of context

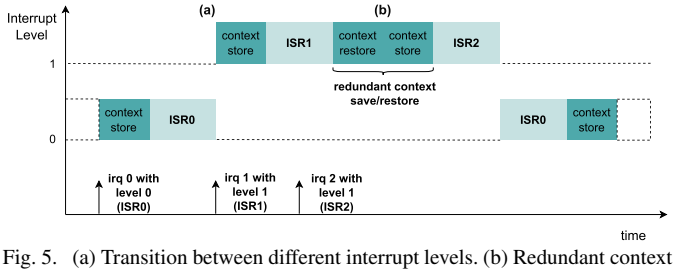


Fig. 5. (a) Transition between different interrupt levels. (b) Redundant context restore of two non-preemptive interrupts (e.g., two interrupts with same level but different priorities) and tail-chaining to optimize it.

switches by interleaving the loading of a new state with the automatic saving HW pushing out the previous register state to memory in the background.

For that, the initial part of the context switch routine changes. At the beginning of the routine, we want to save the current running tasks' state to memory. Instead of manually saving the general-purpose registers, we set up the stack

pointer appropriately before triggering an SW interrupt (write to CLIC's memory map). This will engage the HW mechanism to swap the registers and save them in the background while we can already proceed with the rest of the context switch routine.

Additional RISC-V extensions that introduce more context switching state run contrary to the goals of `fastirq` regarding latencies. While there are no technical limitations with respect to adding more state to `fastirq`, the resulting design would incur a significant increase in area and power. If a specific extension is still required, a dirty bit to make context lazy switchable could help to keep the fast path competitive.

IV. Evaluation

This section gives a functional and quantitative evaluation of the various flavors of the CV32RT. We give an analysis of the functional improvements and argue that these allow more flexible and efficient handling of regular and nested interrupts. Then, we show how the various CV32RT versions perform in terms of interrupt latency and context switch times and finally quantify the overhead these additions incur in terms of area and timing.

A. Feature Comparison

As discussed in Section II, the CLINT is the first standardized RISC-V interrupt controller. It supports pre-emption based on privilege modes (machine, hypervisor, supervisor, user), but the interrupt lines have a hardwired prioritization scheme. For the embedded use cases, the CLINT lacks fine-grained control over interrupt prioritization. The baseline CLIC addresses these weaknesses.

Interrupts can be dynamically assigned a priority and a level. The priorities allow concurrent pending interrupts to be taken in the order preferred by the programmer, while the level information enables pre-emption of same-privilege level interrupts (also called horizontal interrupts). Interrupts that are assigned a higher level can pre-empt lower-level interrupts. In addition to that, a level threshold register per privilege level (`xintthresh`) controls the set of allowed horizontal interrupts by limiting it to those whose level exceeds the given value in the register. This can be useful in critical sections where only a subset of interrupts need to be disabled.

To improve interrupt latencies, interrupt vectoring is supported. Vectoring can be selectively enabled or disabled per interrupt line. This allows for better control of the vector table size. The baseline CLIC does not differ from the regular CLINT in terms of storing and restoring the interrupt context which is fully handled in SW.

The optional `xnxti` extension allows multiple horizontal interrupts to be serviced in sequence without redundant context-restoring operations in between. Reading the `xnxti` CSR yields a pointer to the vector table entry for the next pending and qualifying interrupt, if available, which then allows a direct jump there. This approach loses the latency advantage of HW vectoring by basically running an SW emulation thereof. In addition, the first interrupt has to still pay the full latency cost since `xnxti` does not touch the interrupt context storing part itself.

TABLE II
Feature Comparison of Various Flavors of RISC-V Core-Specific Interrupt Controllers

Configuration	Prioritization by *priv.*— *level* — *priority*	Vectored Interrupts?	Redundant Context Restore
CV32 (CLINT)	✓— ✗— ✗	Both	✗
CV32RT$^{\texttt{CLIC}}$	✓— ✓— ✓	Both	✗
CV32RT$^{\texttt{CLIC}}$ (`xnxti`)	✓— ✓— ✓	✗	✓
CV32RT$^{\texttt{CLIC}}$ (`jalxnxti`)	✓— ✓— ✓	✗	✓
CV32RT$^{\texttt{fastirq}}$	✓— ✓— ✓	✓	✓

Our proposed `fastirq` extension addresses these weaknesses by extending the CLIC base capabilities with a mechanism to lower interrupt latency while still keeping HW vectored interrupts and allowing the skipping of redundant context restore operations.

The discussed differences are summarized in Table II.

B. Functional Performance

1) Measurement Setup: We take measurements by running RTL simulations of the different versions of CV32RT as part of ControlPULP. For the memory subsystem, this means we have single cycle (zero wait state) access to static random access memory (SRAM) (see Section II-A). The memory bank we are using is not contended by other bus masters which is achieved by placing data and instructions into specific private banks. On the system level, there are no additional latencies introduced on interrupt lines between interrupt sources and the CLIC.

2) Interrupt Latency: We measure the HW contributed interrupt latency (as described in Section II-B3) of our `fastirq` extension and compare it to the CV32 and CV32RT variations (standard CLIC, `xnxti`, `jalxnxti`). The interrupt latency is measured as the number of cycles it takes for an interrupt to arrive at the interrupt controller input to the first instruction of an interrupt handler that allows the calling of a C-function. This implies that all caller-save registers need to be saved, which is the general case.

Interrupt handler routines that save the interrupt context in SW can only save the minimum state if the compiler is able to fully inline the handler's function body. Depending on the function and the ABI, this might be fewer than all caller-save registers. To also consider these cases, we measure the interrupt latency in the optimal case, i.e., only one caller-save register needs saving for SW-based interrupt handlers.

Interrupts are injected into the design at the interrupt controller inputs. Each HW configuration has a handwritten optimized interrupt handler that stores all required general-purpose and machine-specific registers for nesting interrupts. We evaluate both the EABI and regular integer ABI of RISC-V. The saved general-purpose registers are all caller-save registers in the respective ABI. This allows for directly calling into C-code functions from interrupt handlers. For our `fastirq` extension, this amount of state is always saved since the HW does not know a priori which caller-save registers need to be saved. Interrupt handler routines that use SW-based mechanisms to save and restore interrupt state can, if the compiler permits, fully inline the handler code and save some caller-save registers.

TABLE III

COMPARISON OF THE MAIN TECHNIQUES FOR OPTIMIZING INTERRUPT CONTEXT AND TASK CONTEXT SAVE/RESTORE WITH NESTED AND NON-NESTED INTERRUPTS EMPLOYED BY INDUSTRY AND ACADEMIA IN THE EMBEDDED AND REAL-TIME APPLICATION DOMAINS

Processor	Interrupt controller	ISA	Interrupt context save/restore acceleration	Task context save/restore acceleration	Back-to-back interrupts acceleration	Open?	Max. #levels	Interrupt latency	Task context switch time	Area Overhead [g]
Industry										
Arm Cortex M4 [20], [25]	NVIC	Arm RISC	Automatic in HW	n.a.	NVIC tail-chaining	✗	256	12	96	n.a.
Arm Cortex R5	VIC/GIC	Arm RISC	n.a.	Register banking	n.a.	✗	n.a.	20	n.a.	n.a.
Infineon AURIX	ICU	TriCore RISC	Automatic in HW	SW based - context save area (CSA)	n.a.	✗	256	10-16	156-162	n.a.
Renesas M32C/80	IC	M32C/80	Dual register bank	Dual register bank	n.a.	✗	8	5 [a]	n.a.	n.a.
Nuclei System Technology	ECLIC	RISC-V	n.a.	n.a.	`jalmnxti`	✗	256	4-6 [b]	n.a.	n.a.
Alibaba's T-Head Xuantie E906	CLIC	RISC-V	n.a.	n.a.	`mnxti`	✓	n.a.	n.a.	n.a.	n.a.
SiFive E21	CLIC	RISC-V	n.a.	n.a.	`mnxti`	✗	16	20 [c]	n.a.	n.a.
ESP32 C3	INTC	RISC-V	n.a.	n.a.	n.a.	✗	15	n.a.	n.a.	n.a.
-	AIA local	RISC-V	-	-	`xtopi`	✗	-	-	-	n.a.
Academia										
Gaitan et al. [26]	MPRA/HSE	RISC	Automatic in HW	Automatic in HW	n.a.	✗	n.a.	1.5 [d]	3 [d]	n.a.
Mao et al. [27]	CLIC	RISC-V	Automatic in HW	n.a.	n.a. [e]	✗	n.a.	13	n.a.	4 kGE [h]
Huang et al. [28]	n.a.	Arm RISC	-	valid-annotated mechanism and semi-shadowing	-	✓	n.a.	n.a.	n.a.	n.a.
Balas et al. [29]	CLINT	RISC-V	-	-	-	✓	-	24	129	10 kGE
This work	**CLIC**	**RISC-V**	**Hybrid [f]**	**Hybrid**	**`emret`**	✓	**256**	**6**	**114**	n.a.

[a] Single high-speed interrupt. Refers to the number of clock cycles from the firing of an interrupt to the execution of the interrupt body. Does not count the saving of the general purpose registers. [30].
[b] Refers to the number of cycles from arrival the arrival of the interrupt to the first instruction fetch of the ISR [31]. Does not count the saving of the interrupt context, including the general purpose registers.
[c] Refers to the number of the clock cycles it takes from the firing of an external interrupt to the time the first instruction in the corresponding interrupt service routine of a C function is executed [18].
[d] Full HW solution. SW scheduling latency becomes negligible as from Figure 1, which justifies the performances in the order of the clock cycle.
[e] Implementation details are omitted.
[f] Combination of background saving/restoring and register banking.
[g] Commercial architectures do not report overheads and costs.
[h] Implemented in a 0.11μm technology node. Total size of the interrupt system.

The results are summarized in Fig. 6. Compared to the baseline CLINT and CLIC which both have about 33 cycles interrupt latency, the `fastirq` extension is able to reduce this down to six cycles. `xnxti` and `jalxnxti` perform even worse (requiring 42 and 35 cycles, respectively) since they both insert additional instructions in the code path between the handler and interrupt event. As a comparison, the Arm Cortex-M4 has an interrupt latency of 12 cycles given a single-cycle memory [24].

3) Redundant Context Restoring: As described in Section II-B, whenever there are *back-to-back* interrupts, the interrupt context restore and store sequence between them can be considered redundant.

In Fig. 6, we show the cost in clock cycles of such sequences. For the baseline CLIC, it takes 68 or 50 cycles when using the integer or embedded ABI, respectively. In this case, the redundant context restoring sequences contains the full interrupt exit code sequence. The latter consists of restoring the interrupt context and the regular interrupt latency. `xnxti` and `jalxnxti` improve upon this situation by checking for pending interrupts and directly jumping to the respective handlers. Note that this is independent of the ABI since no interrupt state is affected. While `xnxti` only returns a pointer to the address of the next handler and thus needs a small code sequence (load, jump, and retry loop), `jalxnxti` fuses these operations into one instruction resulting in saving nine cycles. The `emret` mechanism of `fastirq` works similarly and costs eight clock cycles but is not restricted to non-vectored interrupts. The Arm Cortex-M4 is able to do the same task in six cycles assuming it has access to single-cycle memory [24].

4) Context Switch Time: In Fig. 7, we show the context switch time in number of clock cycles between two FreeRTOS dummy tasks when comparing the baseline CV32RT against CV32RT`fastirq`. All compile time options such as tracing, stack overflow signaling, and the more generic task selection mechanism were turned off to minimize the context switch code. We left out configurations that do not have any impact on context switching. `fastirq` allows us to skip ahead the saving of the general-purpose registers by triggering an SW interrupt as part of the save sequence. The SW interrupt will trigger the `fastirq` mechanism, which starts saving the general-purpose registers to memory. By making sure not to use registers that are still being saved by the background-saving mechanism (as described in Section III-C4), we can save up to 31 cycles (−19%) for a context switch when using the I-extension, and 16 cycles (−12%) for the E-extension, respectively. The FreeRTOS website [20] claims context switches as low as 96 cycles for a Cortex-M4 implementation.[3]

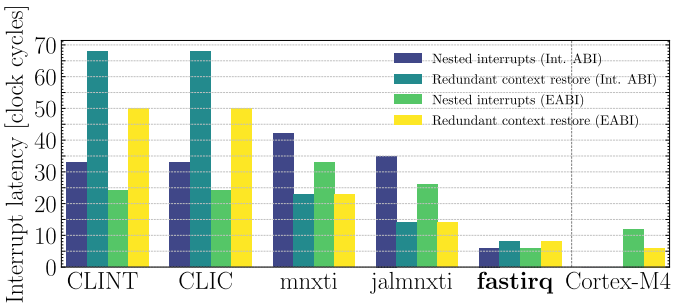


Fig. 6. Interrupt latency of various flavors of RISC-V core-specific interrupt controllers for interrupt handlers that support nesting and the calling of C-functions within it (i.e., saving/restoring state following the C-ABI). As a comparison, we mention the generic Cortex-M4 core, assuming it has single-cycle access to memory. Note that Arm Cortex-M only has 16 core registers.

C. Implementation

We synthesize CV32RT as part of the ControlPULP platform using Synopsys Design Compiler 2022.03, targeting GlobalFoundries 12LP FinFet technology at 500 MHz, TT corner, and 25 °C. One gate equivalent (GE) for this technology equals 0.121 μm^2.

[3]This can be seen as a lower bound as on some SoCs such as the STM32L476RG we observed higher latencies due to memory access stalls and other implementation choices in the memory subsystem.

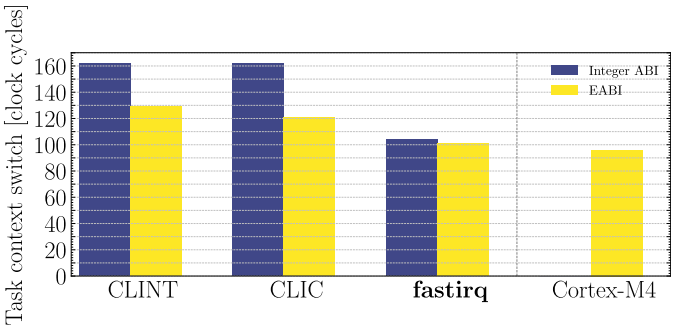


Fig. 7. Average context switch time in FreeRTOS for two tasks of various flavors of CV32RT. The RISC-V E-Extension reduces the available general-purpose registers from 32 to 16, consequently lowering the context switch state that needs to be saved and restored.

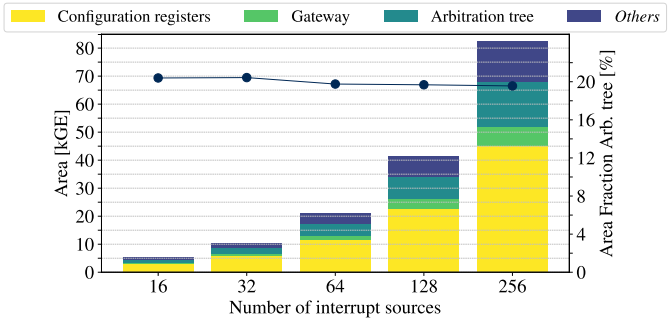


Fig. 8. RISC-V CLIC area breakdown at varying numbers of interrupt sources.

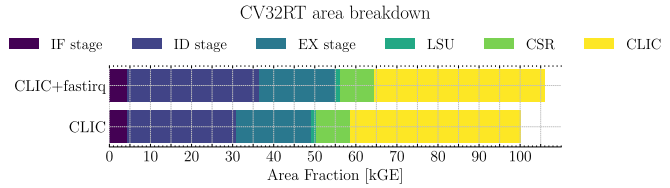


Fig. 9. Area overhead of CV32RT in the two main configurations. The overhead of `fastirq` in CV32RT$^{\texttt{fastirq}}$ core results in a minimal 10% area increase concentrated around the ID stage. The design has been synthesized in GF12LP technology (500 MHz, TT corner, 25 °C, 0.8 V, super low V_T standard cells).

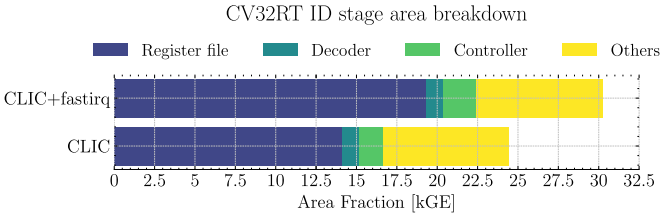


Fig. 10. Area breakdown of CV32E40P^{RT} *ID stage* with the proposed HW extensions.

Fig. 8 reports the area breakdown of the CLIC implemented in the proposed with different interrupt sources. As shown in Fig. 8, for each scenario, more than half of the resources implement the configuration registers required to control the CLIC, whose size linearly increases with the number of input interrupts. In particular, each interrupt is associated with one 32-bit register incurring an area overhead of about 176 GE. The remaining area implements the gateway and binary tree arbitration logic at the core of the CLIC working principle, as discussed in Section III-B, and additional housekeeping control logic that scales linearly with the number of interrupt sources. Fig. 8 also shows that the fraction of the design occupied by the arbitration tree is kept constant when increasing the number of sources. While it is apparent that this design incurs a larger area overhead compared to traditional RISC-V CLINT [14], the gain in flexibility enables a broader application scope with time-critical systems.

Fig. 9 reports the area breakdown comparison of CV32RT with baseline CLIC (including `mnxti` and `jalmnxti` CSRs) and `fastirq` extensions with 256 input interrupts. From Table II, CV32RT$^{\texttt{fastirq}}$ incurs a 10% overall area increase compared to baseline CLIC only. While other HW blocks of the core remain primarily unaffected, the instruction decode (ID) stage incurs an area overhead of 21% compared to CV32RT$^{\texttt{CLIC}}$. The ID stage is the HW block where the additional registers and the automatic stacking/unstacking logic are localized. A breakdown of the ID stage is shown in Fig. 10. We notice that the additional storage space for automatic context save and restore in HW increases the area of the RF by about 36% in the proposed implementation. Similarly, the logic for managing the shadow registers accounts for an overhead of 40% on the baseline *ID stage* controller. The HW overhead coming from the additional `emret` instruction discussed in Section IV is negligible. Nevertheless, the increased size of the ID stage trades off the benefits of: 1) a simplified programming model that moves several SW operations in HW and 2) significantly lowered interrupt latency than standard RISC-V while not impacting the critical path of the base core design. Finally, time-critical systems shift design priorities from area efficiency to safety, security, and reliability.

V. Related Work

In this section, we describe the leading solutions to optimize handling asynchronous events in state-of-the-art (SOTA) embedded and real-time MCUs. We first differentiate between existing PLICs and CLICs as introduced in Section II-C. Then, we focus on the latter and discuss solutions across various platforms in industry and academia, addressing: 1) interrupt context save/restore techniques; 2) context switch techniques; and 3) dedicated strategies to optimize redundant context restore with back-to-back interrupts. Table III summarizes the overview. For the interrupt latency, we assume that the definition presented in Section II-B and explicitly provide references to different variants adopted by SOTA when needed in Table III.

A. Platform-Level Interrupt Controllers

Controllers that belong to this category, such as the RISC-V PLIC and advanced PLIC (APLIC) for wire-based interrupt communication and RISC-V incoming message signaled interrupt controller (IMSIC) for message-signaled interrupt communication, are not designed to distribute time-critical interrupts to the running HARTs. Conversely, Arm's generic interrupt controller (GIC) redistributes incoming asynchronous events as either non-critical (`IRQ`) or critical interrupts (fast `IRQ`, or `FIQ`). The latter reduces the execution time of the

interrupt handler through a dedicated register bank, with up to eight registers employed to minimize context switching.

B. Core-Local Interrupt Controllers

CLICs are often specialized in providing fast interrupt-handling capabilities in real-time embedded application domains.

1) Interrupt Context Save/Restore Acceleration: Several designs automatically save and restore the interrupt context directly in HW. This allows the core to start fetching the jump address from the interrupt vector table simultaneously during context save and embed context restore within the return instruction. A reference example in the field is the Arm Cortex-M series integrating the nested vectored interrupt controller (NVIC). It implements a state machine [32] that performs caller-save register stacking in the background. Likewise, upon returning from the ISR, the HW encodes in the link register a value (`EXC_RETURN`), which notifies the core to start unwinding the stack to return to normal program execution. Analogously, in Infineon AURIX MCU-class TriCore family's interrupt control unit (ICU) [33], [34], [35], the context of the calling routine is saved in memory autonomously while restoring the context is embedded in the `RET` instruction and happens in parallel with the return jump [36]. From academia, Mao et al. [27] are the first to propose extensions for the RISC-V CLIC. Though implementation details are omitted, the interrupt handling is enhanced with automatic stacking in HW that benefits from the core's Harvard architecture and simultaneous data and instruction memory access.

Albeit automatic interrupt context save/restore reduces SW housekeeping overhead before and after handling the interrupt routine, it only partially addresses the acceleration of the complete task context switch.

2) Task Context Save/Restore Acceleration: Register banking is a technique adopted by several architectures to swap a task's context without pushing/popping register values to the stack at the cost of an additional area overhead in the design. As discussed above, in Arm designs, this is the case of HW register banking in PLICs (GIC). A similar approach is implemented in the Renesas M32C/80 series [30]. A dual register bank allows quickly swapping the context without saving/restoring to/from the stack, as the second register bank is reserved for high-speed interrupts. The Aurix family instead implements an SW managed solution, featuring a specific organization of the context layout in the system memory based on context save area (CSA) chained in a linked list fashion [36]. A more complex approach based on the intuition that often the content of some register remains untouched after a context switch is introduced by Huang et al. [28], which adopts a valid-based mechanism in HW to block context switch on selected registers, reducing register movement by almost 50%. This feature is combined with semi-shadowing, similar to register banking but using the top (*flip*) and bottom (*flop*) halves of the RF as RF copies. The latter approach is similar to leveraging RISC-V's RV32E base instruction set by re-using the lower 16 architectural registers of the RF, as proposed in this work. The evaluation in [28] lacks HW implementation and area overhead assessment but reduces context switching overhead by 24% on the DSPstone benchmark.

These approaches trade-off HW- and SW-induced latencies when handling asynchronous events, as depicted in Fig. 1. Gaitan et al. [26] propose a full HW solution based on a hardware scheduling engine (HSE) that directly attaches interrupts to running tasks without the need for a specialized interrupt controller. While this approach allows lowering interrupt latency and task context switches dramatically, as reported in Table III, it lacks flexibility and its area overhead grows for a high number of tasks, as it requires replicating hardware resources per task.

Despite exceeding the scope of real-time execution, we mention that fast context switching is often required in architectures such as superscalar central processing units (CPUs) or to hide latency in graphic processing units (GPUs) [37], [38]. Besides highlighting the difference in memory footprint between such architectures (MB size RFs) and low-end embedded microcontrollers targeted in this work, such works typically adopt *RF caching* rather than register shadowing or banking for performance reasons, i.e., lower access latency to the RF and higher thread-level parallelism (TLP), respectively. In the real-time application scenario of this work, such techniques may deteriorate the system's predictability and thus are not considered.

3) Optimizations for Non-Nested Interrupt Handling: Redundant context restore with non-nested interrupts is addressed by chaining two back-to-back interrupts and bypassing the superfluous restore/save operation. RISC-V addresses this scenario by adding the `xnxti` CSR with SW-managed interrupt service loops as part of the CLIC specifications and finds early commercial examples with [31]. Nuclei System Technology ECLIC [39] extends traditional `xnxti` with a novel CSR for machine privilege mode, `jalmnxti` [18], already discussed in this work in Section IV. Among MSI solutions, RISC-V AIA without APLIC has a similar approach to `mnxti` with the `xtopi` CSR that reports the highest-priority, pending, and enabled interrupt for a specific privilege mode, allowing both late arrival and redundant context restore mechanisms.

While the presented solutions are effective in optimizing context, save/restore with HW and SW cooperation, they lack: 1) a cohesive approach to address both interrupt context and task context switch acceleration and 2) none of the existing RISC-V-based approaches can close the gap with well-established industry vendors.

This work proposes a combination of the background-saving with a register banking approach. Interrupt context save/restore especially takes advantage of the former by deferring those operations to the HW. This allows execution of the interrupt handler before all interrupt state has been pushed to memory, lowering interrupt latency. A task's context switch benefits from quickly transferring the suspended context to the dedicated register bank while already restoring the next task to be executed.

To address redundant context restore with back-to-back interrupts, we propose a modified version of the return

instruction in machine mode (i.e., `emret`), which is able to skip redundant context saving and restoring sequences by directly jumping to the next available interrupt handler. This solution works for both vectored and non-vectored interrupts.

VI. Conclusion

In this work, we present `fastirq`, a fast interrupt extension for RISC-V embedded systems. We implement the extension on CV32RT, a 32-bit, in-order, single-issue core designed with the RISC-V CLIC fast interrupt controller. With our design, we can achieve interrupt latencies of six clock cycles and efficient back-to-back interrupt handling in 12 cycles which is as low as the fastest available approaches currently implemented in the RISC-V landscape, fully open-source, and competitive against closed-source and proprietary commercial solutions. Furthermore, we improve task context switch times in FreeRTOS to 104 clock cycles using `fastirq`, which is 20% faster than an SW-only approach. Some research directions involving the proposed extension that the authors consider for future work are analyzing `fastirq`'s impact on timing channels or its integration with different RISC-V extensions.

References

[1] C. Rochange, S. Uhrig, and P. Sainrat, *Time-Predictable Architectures* (FOCUS—Computer Engineering Series). Hoboken, NJ, USA: Wiley, 2014. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1848215932.html

[2] L. M. Pinho et al., *High-Performance and Time-Predictable Embedded Computing*. Wharton, TX, USA: River, 2018.

[3] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time Linux kernel: A survey on PREEMPT_RT," *ACM Comput. Surv.*, vol. 52, no. 1, pp. 1–36, Feb. 2019, doi: 10.1145/3297714.

[4] M. Liu, D. Liu, Y. Wang, M. Wang, and Z. Shao, "On improving real-time interrupt latencies of hybrid operating systems with two-level hardware interrupts," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 978–991, Jul. 2011.

[5] P. Mantegazza, E. L. Dozio, and S. Papacharalambous, "RTAI: Real time application interface," *Linux J.*, vol. 2000, no. 72, p. 10, Apr. 2000.

[6] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proc. IEEE*, vol. 82, no. 1, pp. 55–67, Jan. 1994.

[7] D. Kleidermacher. (2005). *Minimizing Interrupt Response Time*. [Online]. Available: http://web.engr.oregonstate.edu/~traylor/ece473/pdfs/minimize_interrupt_response_time.pdf

[8] J. Valvano, *Introduction to Embedded Systems*. Scotts Valley, CA, USA: CreateSpace, Aug. 2016.

[9] Y. Huang, L. Shi, J. Li, Q. Li, and C. J. Xue, "WCET-aware re-scheduling register allocation for real-time embedded systems with clustered VLIW architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 1, pp. 168–180, Jan. 2014.

[10] X. Zhou and P. Petrov, "Rapid and low-cost context-switch through embedded processor customization for real-time and control applications," in *Proc. 43rd Annu. Conf. Design Autom. (DAC)*. New York, NY, USA: Association for Computing Machinery, 2006, p. 352, doi: 10.1145/1146909.1147001.

[11] I. Behnke, L. Pirl, L. Thamsen, R. Danicki, A. Polze, and O. Kao, "Interrupting real-time IoT tasks: How bad can it be to connect your critical embedded system to the Internet?" in *Proc. IEEE 39th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Nov. 2020, pp. 1–6, doi: 10.1109/IPCCC50635.2020.9391536.

[12] F. Rehm et al., "The road towards predictable automotive high–Performance platforms," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 1915–1924.

[13] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Dept. EECS, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-146, Aug. 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[14] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual volume II: Privileged architecture version 1.9," Dept. EECS, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-129, Jul. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html

[15] *RISC-V 'SMCLIC' Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extension*. Accessed: Jul. 18, 2023. [Online]. Available: https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc

[16] M. Gautschi et al., "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017.

[17] D. Schiavone. *OpenHW Group CV32E40P User Manual*. OpenHW Group. Accessed: Jun. 2, 2023. [Online]. Available: https://cv32e40p.readthedocs.io/en/latest/

[18] Nuclei System Technology. (2021). *Nuclei ISA Spec*. [Online]. Available: https://doc.nucleisys.com/nuclei_spec/isa/introduction.html

[19] A. Ottaviano et al., "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," *Int. J. Parallel Program.*, Feb. 2024, doi: 10.1007/s10766-024-00761-4.

[20] R. Barry. (2023). *FreeRTOS: Real-Time Operating System for Microcontrollers*. Real Time Engineers Ltd. [Online]. Available: https://www.freertos.org/index.html

[21] C.-M. Lin, "Nested interrupt analysis of low cost and high performance embedded systems using GSPN framework," *IEICE Trans. Inf. Syst.*, vol. E93-D, no. 9, pp. 2509–2519, 2010.

[22] RISC-V Internationa. (2023). *RISC-V Platform-Level Interrupt Controller Specification*. [Online]. Available: https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic-1.0.0.pdf

[23] J. Yiu, *The Definitive Guide to ARM Cortex-M3 Cortex-M4 Processors*, 3rd ed. Boston, MA, USA: Newnes, 2013.

[24] ARM. (2020). *Cortex-M4 Tech. Reference Manual*. [Online]. Available: https://developer.arm.com/documentation/100166/0001/

[25] J. Yiu. (2012). *A Beginner's Guide Interrupt Latency—Interrupt Latency Arm Cortex-M Processors*. ARM. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/beginner-guide-on-interrupt-latency-and-interrupt-latency-of-the-arm-cortex-m-processors?pifragment-22714=2

[26] V. G. Gaitan, N. C. Gaitan, and I. Ungurean, "CPU architecture based on a hardware scheduler and independent pipeline registers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 9, pp. 1661–1674, Sep. 2015.

[27] B. Mao, N. Tan, T. Chong, and L. Li, "A CLIC extension based fast interrupt system for embedded RISC-V processors," in *Proc. 6th Int. Conf. Integr. Circuits Microsystems (ICICM)*, Oct. 2021, pp. 109–113.

[28] C.-W. Huang, K.-Y. Hsieh, J.-J. Li, and J. K. Lee, "Support of paged register files for improving context switching on embedded processors," in *Proc. Int. Conf. Comput. Sci. Eng.*, 2009, pp. 352–357.

[29] R. Balas and L. Benini, "RISC-V for real-time MCUs—Software optimization and microarchitectural gap analysis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 874–877.

[30] Renesas. (2023). *Hardware Manual—RENESAS MCU M16C*. [Online]. Available: https://www.renesas.com/us/en/document/mah/m32c87-group-m32c87-m32c87a-m32c87b-hardware-manual

[31] SiFive Inc. (2021). *SiFive E21 Core Complex Manual*. [Online]. Available: https://sifive.cdn.prismic.io/sifive/7c22c2ec-8af4-4b6c-a5fe-9327d91e7808_e21_core_complex_manual_21G1.pdf

[32] STMicroelectronics. (2020). *STM32L5—NVIC*. [Online]. Available: https://www.st.com/content/ccc/resource/training/technical/product_training/group1/61/35/d2/07/34/6f/4e/83/STM32L5-System-Nested_Vectored_Interrupt_Control_NVIC/files/STM32L5-System-Nested_Vectored_Interrupt_Control_NVIC.pdf/_jcr_content/translations/en.STM32L5-System-Nested_Vectored_Interrupt_Control_NVIC.pdf

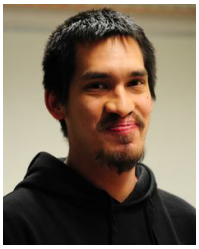
[33] Infineon Technologies AG. A Fast Powertrain Microcontroller. [Online]. Available: https://old.hotchips.org/wp-content/uploads/hc_archives/hc16/3_Tue/6_HC16_Sess7_Pres1_bw.pdf

[34] Infineon Technologies AG. (2014). *TC27x D-Step, 32-Bit Single-Chip Microcontroller*. [Online]. Available: https://hitex.co.uk/fileadmin/uk-files/downloads/ShieldBuddy/tc27xD_um_v2.2.pdf

[35] Infineon Technologies AG. (2012). *TriCore V1.6, Core Architecture*. [Online]. Available: https://www.infineon.com/dgdl/tc1_6__architecture_vol1.pdf?fileId=db3a3043372d5cc801373b0f374d5d67

[36] Infineon Technologies AG, "Task context switching RTOS," U.S. Patent 7 434 222 B2, Oct. 2008.

[37] H. Zeng and K. Ghose, "Register file caching for energy efficiency," in *ISLPED Proc. Int. Symp. Low Power Electron. Design*, Oct. 2006, pp. 244–249.
[38] M. Sadrosadati et al., "Highly concurrent latency-tolerant register files for GPUs," *ACM Trans. Comput. Syst.*, vol. 37, nos. 1–4, pp. 1–36, Jan. 2021, doi: 10.1145/3419973.
[39] Nuclei System Technology Co. Ltd. *ECLIC Unit Introduction*. Accessed: Aug. 4, 2023. [Online]. Available: https://doc.nucleisys.com/nuclei_spec/isa/eclic.html

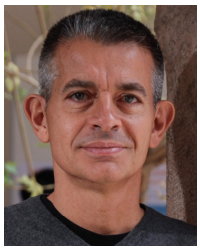


Robert Balas (Graduate Student Member, IEEE) received the B.Sc. and M.Sc. degrees in electrical engineering and information technology from ETH Zürich, Zürich, Switzerland, in 2015 and 2017, respectively, where he is currently working toward the Ph.D. degree at the Digital Circuits and Systems Group of Prof. Benini.

His research interests include real-time computing, compilers, and operating systems.

Alessandro Ottaviano (Graduate Student Member, IEEE) received the B.Sc. degree in physical engineering from the Politecnico di Turino, Turin, Italy, in 2018, and the joint M.Sc. degree in electrical engineering from the Politecnico di Turino, Grenoble INP-Phelma, Grenoble, France, and EPFL Lausanne, Lausanne, Switzerland, in 2020. He is currently working toward the Ph.D. degree at the Digital Circuits and Systems Group of Prof. Benini, ETH Zürich, Zürich, Switzerland.

His research interests include real-time computing, power management of HPC processors, and energy-efficient processor architecture.



Luca Benini (Fellow, IEEE) received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 1997.

He currently holds the Chair of the Digital Circuits and Systems Group, ETH Zürich (ETHZ), Zürich, Switzerland, and is also a Full Professor at the Università di Bologna, Bologna, Italy.

Dr. Benini is a fellow of the ACM and a member of the Academia Europaea. He was a recipient of the 2023 IEEE CS E.J. McCluskey Award.