mathematics

MDPI

Article

Designing a Custom CPU Architecture Based on Hardware RTOS and Dynamic Preemptive Scheduler

Ionel Zagan [1,2,*] and Vasile Gheorghiță Găitan [1,2,*]

[1] Faculty of Electrical Engineering and Computer Science, Stefan cel Mare University, 720229 Suceava, Romania
[2] Integrated Center for Research, Development and Innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for Fabrication and Control (MANSiD), Stefan cel Mare University, 720229 Suceava, Romania
* Correspondence: zagan@usm.ro (I.Z.); vgaitan@usm.ro (V.G.G.)

Abstract: The current trend in real-time operating systems involves executing many tasks using a limited hardware platform. Thus, a single processor system has to execute multiple tasks with different priorities in different real-time system (RTS) work modes. Hardware schedulers can greatly reduce event trigger latency and successfully remove most of the scheduling overhead, providing more computing cycles for applications. In this paper, we present a hardware-accelerated RTOS based on the replication of resources such as program counters, general purpose registers (GPRs) and pipeline registers. The implementation of this new concept, based on real-time event handling implemented in hardware, is intended to meet the current rigorous requirements imposed by critical real-time systems. The most important attribute of this FPGA implementation is the time required for task context switching, which is only one clock cycle or three clock cycles when working with the atomic instructions used in the case of inter-task synchronization and communication mechanisms. The main contribution of this article is its focus on mutexes and the speed of response associated with related events. Thus, fast switching between threads is also validated, considering the handling of events in the hardware using HW_nMPRA_RTOS (HW-RTOS). The proposed architecture implements inter-task synchronization and communication mechanisms with high performance, improving the overall response time when the mutex or message is expected to relate to a higher-priority task.

Keywords: embedded systems; field-programmable gate arrays (FPGAs); real-time mutex event handling; task scheduling

MSC: 68N25

Citation: Zagan, I.; Găitan, V.G. Designing a Custom CPU Architecture Based on Hardware RTOS and Dynamic Preemptive Scheduler. *Mathematics* **2022**, *10*, 2637. https://doi.org/10.3390/math10152637

Academic Editors: Mihai-Victor Micea, Alex Doboli, Daniel-Ioan Curiac, Cristina Sorina Stângaciu and János Sztrik

Received: 22 June 2022
Accepted: 26 July 2022
Published: 27 July 2022

1. Introduction

In order to improve performance, hardware architects have aimed to increase computer clock frequency; moreover, they have enhanced the instructions per cycle (IPC) coefficient by increasing the number of instructions that are completed in a clock cycle. For this to be accomplished, it is necessary to implement multi-stage assembly lines and then execute multiple instructions in parallel, thus obtaining a superscalar architecture. The main problem that arises from using a pipeline or multiple pipelines is maximizing the use of each stage of the assembly line [1,2]. To address this problem, architects have proposed and created complex mechanisms for executing instructions in a different order, thus increasing the complexity of the entire processor while maintaining power consumption within acceptable limits. An alternative to this problem, which allows for a high degree of parallelization at the instruction level, is a technique called multithreading. In essence, the instruction flow is divided into several fluxes named threads, so that these threads can be executed in parallel. A variety of multithreading projects based on different architectures [3] have been realized and implemented in experimental [4] and even commercial projects.

Programming languages often hide the instruction set architecture (ISA) and do not pass the predictability test because they do not express the temporal properties of the assembly language used. In addition, real-time operating systems (RTOSs) do not pass the predictability test because they hide the complexity of concurrent synchronization and communications between tasks, thus hiding the temporal delays generated by this orchestration. As a result, it is often desirable to determine real-time behavior using a test and evaluation software benchmark. Task contexts' saving and restoring operations are also important issues when working with real-time kernels. Information regarding the tasks of an operating system is stored in data structures called task control blocks (TCBs). These TCBs must contain all the necessary parameters for creating the tasks and the necessary information for managing them.

RTOSs are primarily used for their accelerated response capability. A real-time scheduler is a program unit that controls execution and temporary preemption and completes the execution of some program modules based on a predefined algorithm to meet the required time constraints. Hardware schedulers and HW RTOSs [5] are designed to relieve the processor from task scheduling overhead, enabling a worst-case execution time (WCET) to be specified [6]. The preemptive schedulers introduce fluctuations in task execution times, degrading the performance of the RTS. A major drawback of non-preventive software scheduler implementations is that they introduce an additional blocking factor for high-priority tasks. Nevertheless, there are several important advantages when adopting this type of scheduler. Current benchmarks assess the accuracy of running programs without considering how long it takes to execute certain instruction sequences.

This paper begins with a brief introduction, and then Section 2 compares the proposed implementation with that of other similar projects. Sections 3 and 4 present the real-time event handling based on a hardware RTOS architecture and the integrated hardware scheduler. Section 5 describes the implementation of the proposed architecture using the Virtex-7/Nexys 4 DDR development kit, also presenting the resource requirements for implementing the processor using Verilog HDL. Sections 6 and 7 present the discussions, conclusions and future directions of research.

2. Related Work

The authors of the present paper aimed at a realistic comparison between different CPU architecture implementations. The XMOS processor presented by May in [7] has a scalable architecture, so it can use the entire central processing unit even if the number of active execution threads is less than four. The new XMOS architecture allows designers to build systems with multiple Xcore kernels connected. Communication between Xcore cores from the same chip or different chips is performed using messages sent through point-to-point communication links, ensuring the predictive execution of concurrent programs. The cores interact with other external devices via integrated ports. Therefore, the XMOS architecture can be used successfully to build multi-core systems, dedicated boards or distributed systems. The processor core proposed in [8] is composed of two distinct pipelines. The first one is dedicated to a single hard real-time (HRT) execution thread, and the second pipeline is dedicated to non-HRT (NHRT) execution threads. As can be seen from the example presented by the authors, in a quad-core version, each core is composed of four hardware slots. Thus, each core can simultaneously execute an HRT thread and three NHRT threads. The HRT thread is assigned the highest priority, being isolated from the other NHRT threads in the core through the real-time scheduler. The threads' priorities are fixed, and round-robin is the chosen scheduling scheme. Each kernel is composed of two scratchpad memories, one for data and the other for instructions (D-ISP and DSP) and ensures data integrity by individually assigning a subset from a bank cache to each task [9]. To minimize interference between tasks, the authors of this paper propose using an analyzable real-time memory controller. The disadvantage of this project is the increased resource requirements and rigidity because every core can only have one HRT thread and an arbitrary number of NHRT threads.

In [10], Clemente et al. present a special implementation of a run-time hardware scheduler designed for reconfigurable systems. The authors designed and validated a run-time scheduler that operates with task graphs. The task graphs are analyzed at design time and the information extracted is used at run-time in order to obtain near-optimal scheduling operations. The performance of implementing this scheduler in hardware applies to all optimization techniques while introducing a delay of only a few clock cycles. The experimental results presented in this paper prove that the proposed scheduler outperforms conventional run-time schedulers based on as-soon-as-possible techniques. In addition, our scheduler provides efficient management of the execution of task graphs for reconfigurable multitasking systems, which can significantly improve the performance of the system and also reduce energy consumption. In other architectures [11], architects focus on designing flexible processors for embedded applications, reducing energy consumption and improving speed and design time. Vermeulen et al. proposed a novel hybrid CPU architecture [11] that allows the implementation of time-critical functionality on a custom accelerator, thus preserving the flexibility of the platform implementation. To solve the data transfer and storage bottleneck for multimedia applications, a customized memory architecture is shared with the flexible component. The MIPS processor core proposed by Gschwind et al. in [12] represents an FPGA application-specific processor prototype designed for embedded applications. The project was developed using the MIPS-I ISA architecture and the VHDL hardware description language. The authors design a reconfigurable MIPS processor core to support hardware/software co-evaluation of instruction sets for design space exploration. Therefore, we can see that with the arrival of high-density FPGA devices, prototyping has become accessible to the designers of integrated applications built around application-specific processors. Besides the MicroBlaze [13] and Amber 23 processors [14,15] (Figure 1), all processor implementations taken into consideration use complete or partial resource multiplication for 4 sCPUs/tasks/threads/contexts. The results obtained by the designers are related to the architecture of the implemented processor [16–19], with a particular impact on the data presented in the graph in Figure 1.

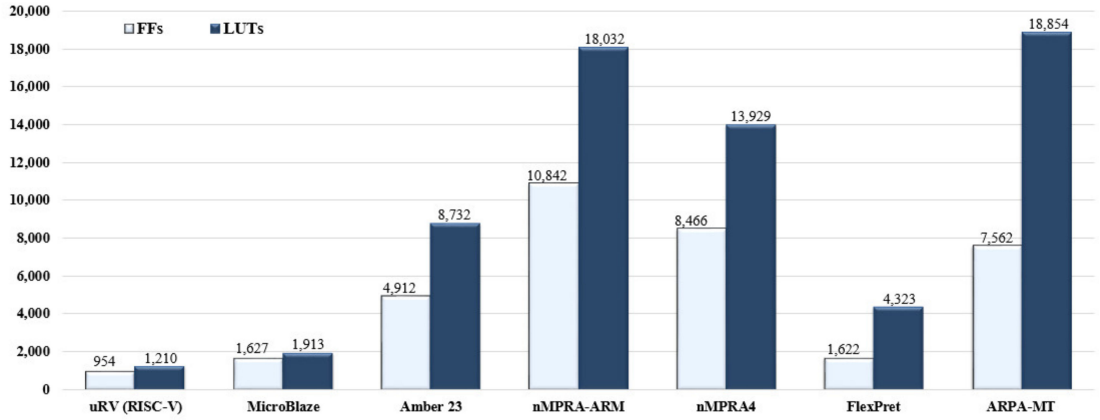


Figure 1. FPGA resources used for implementing the processors described in the literature (uRV(RISC-V) [4], MicroBlaze [13], Amber 23 [14], nMPRA-ARM [19], nMPRA4 [6], FlexPret [17], ARPA-MT [18]).

3. Real-Time Event Processing Based on Hardware RTOS Architecture Support

The proposed concept is based on a five-stage pipeline assembly line and the multiplication of pipeline registers, program counter, GPR and each memory element on the datapath (such as flip-flops associated with the condition indicators, the sequential machine in the division unit and the flip-flops in COP0, etc.). An instance of the CPU will be referred to as a semi-CPU (sCPUi for task i). Such a hardware instance comprises its program counter register, pipeline registers, GPR and its control registers. Therefore, the sCPUi runs the instructions of task i ($i = 0, \ldots, n-1$) based on the nMPRA concept (multipipeline register architecture, where n is the degree of multiplication) [6]. The HW_nMPRA_RTOS

project (nMPRA + nHSE) presented in this paper has been designed using the MIPS32 Release 1 ISA. MIPS (microprocessor without interlocked pipelined stages) provides the user with a system of coprocessors for extending the functionality of the basic CPU. Coprocessor 2 (COP2) is available to the user, so this research project consists of a system-on-chip (SoC) implementation of nMPRA and nHSE at the level of MIPS32 COP2, with the scheduler registers being explained in detail in the specifications of the nMPRA processor. The requirements of a hardware-implemented RTOS must comprise (but not be limited to) the following: guaranteeing a high level of real-time performance, quick interrupt (event) response, fast task context switch and application programming interface (API) execution, increased predictability and much lower CPU resource usage. In the proposed architecture, named HW_nMPRA_RTOS, each task has an associated timer that can be configured to generate an event when the time allowed for that task is nearing completion. Each hardware block is composed of three timers, one representing the recurrence period of the task that might generate an event at the specified time frame, and the following two timers representing the implementation of two deadlines. The first one is a soft deadline that can be an alarm, whereas the second is a hard deadline, equivalent to a fault.

The main contribution of this paper is a novel hardware scheduler that includes support for RTOS kernels (HW_nMPRA_RTOS), aimed at achieving worst-case latencies in the order of nanoseconds for the FPGA-based project. As a derivative contribution, we have implemented the structure of a hardware RTOS for MIPS32-COP2 in order to add predictive scheduler behavior and minimize kernel latency. The implementation of HW_nMPRA_RTOS comprises a real-time hardware operating system that achieves excellent performance at a low cost compared to conventional RTOS software implementations. Compared to simultaneous multithreading (SMT) processor designs, the current implementation only executes one sCPUi at any one time. HW_nMPRA_RTOS cannot run both sCPU0 and sCPU3 simultaneously because there is no multiplied ALU unit or condition testing unit. We have a single data memory and a sign extension unit (all combinational parts), etc.

The application field includes many automotive and robotics applications for which the overhead of commercial off-the-shelf CPUs is too high. Considering that FPGA vendor-provided cores, such as Microblaze and Nios, are not portable and have no sources available either, our HW_nMPRA_RTOS architecture favors an FPGA footprint and determinism of execution over performance, with code and data stored in the internal FPGA RAM. The hardware real-time event handling module validated in this paper is based on the CPU resource multiplication concept patented in Germany, Munich (DE202012104250U1) [20].

The development of this circuit in Verilog HDL transforms the processor code representing the RTL level into a variety of other equivalent visual representations. Thus, using the RTL Netlist, Schematic and Graphical Hierarchy options, the processor can be viewed at different design stages. These options also provide the ability to debug and verify by using the cross-select property. The process of synthesizing the project represents the next step in the FPGA implementation. It involves transforming the project from RTL into logical gate representation. This means that the result from the Verilog code, together with the standard UNISIM libraries, is the non-logical gateway, containing primitives such as flip-flops (FFs) or look-up tables (LUTs). The next step in the validation of the proposed processor is the implementation of the project in the Virtex-7 FPGA circuit. This process consists of placement and routing operations, which, along with the corresponding algorithms, put the netlist elements into the FPGA circuit and connect them so that all requirements are met. This step may be quite slow, especially when using hardware debugging tools such as ChipScope Analyzer.

It is a known fact that the MIPS architecture assigns COP2 for user-specific implementations. COP2 has its own register file (RF), but it is transparent to the programmer due to the real-time event handling module. There are six COP2 instructions that are used to access local and global nHSE registers. The data transfer between nHSE, GPR and data memory is achieved by using the instructions implemented at the level of COP2, namely, CFC2 (copy

control word from coprocessor 2 to GPR, opcode = 010010), CTC2 (copy control word from GPR to coprocessor 2, opcode = 010010), MFC2 (move word (mr) from coprocessor 2 to GPR, opcode = 010010), MTC2 (move word (mr) from GPR to coprocessor 2, opcode = 010010), LWC2 (load word from data memory to COP2 from data memory, opcode = 110010) and SWC2 (store word from COP2 to data memory, opcode = 111010). Figure 2 shows the datapath effect of COP2 instruction execution, based on the nHSE module (hardware scheduler engine for *n* threads). Thus, the block diagram indicates the datapath corresponding to the LWC2 instruction type that loads a word from the data memory into the hardware scheduler register, and the SWC2 instruction stores a word (control register abbreviated further as cr) from the hardware scheduler in the data memory. By executing a code sequence written specifically for validating the real-time event handling unit, the datapath presented in Figure 2 will be tested, even in the case of a hazardous situation. Figure 2 is an architectural diagram, and the Vivado implementation differs essentially from this architecture both in the way the code is written and in the internal structure of the FPGA circuit, optimally used by the Verilog compiler.

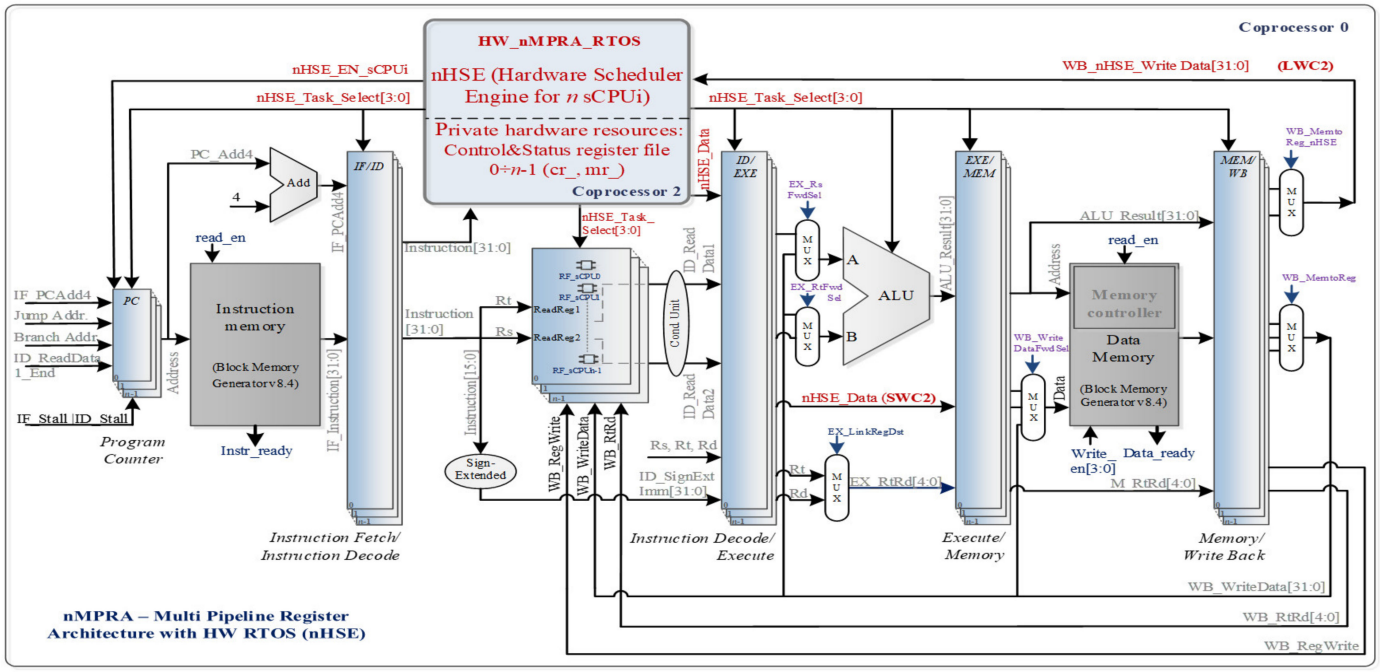


Figure 2. HW_nMPRA_RTOS microarchitecture with nHSE representing an RTOS implemented in hardware based on LWC2- and SWC2-type instructions.

By the validation of the COP2 dedicated instructions, all registers associated with the hardware scheduler can be written or read in GPR space. Changing the number of sCPUi, interrupts, or mutexes does not entail the multiplication of instructions dedicated to the real-time scheduler. The multiplication of monitoring (mr) and control registers at the level of COP2 contributes to the total number of resources required to implement this processor architecture. The advantage of this is the existence of separate contexts for each sCPUi, thus resulting in an additional speed boost that comes from eliminating the need to save and restore parameters on the stack. The hardware scheduling unit is designed to activate only one of the *n* sCPUis at a given moment. Efficiency is another feature that RTS needs to acquire so the real-time scheduler can satisfy all requests for task execution of a scalable system with limited hardware resources. In this context, it can be said that an RTS must be robust and safe, even in those situations where requirements reach their maximum points. The purpose of this project is to ensure the proper functionality of the process, even if the results produced after the deadlines are still used in some RTSs. Although tolerance to errors is another important aspect of the RTS, the scheduler must not allow the

existence of unpredictable situations that may affect the safety of a human operator or even of the product beneficiaries. Compared to a general-purpose operating system, an RTOS typically uses round-robin scheduling to ensure the accuracy of real-time task processing. The earliest deadline first (EDF) algorithm can also be used to schedule an independent set of preemptive and aperiodic tasks that run on a single-core system. For this, each task τi is characterized by a WCET marked with Ci, a deadline Di, an execution period Ti and a Pi priority, used to select which of the ready-to-run n tasks can be scheduled. A lower value for Pi represents a higher priority for the respective task, as follows: $\forall i \mid 1 \leq i < n$: Pi < Pi + 1. The smaller number of task context switch changes in EDF is a direct consequence of assigning dynamic Pi priorities according to the earliest deadline, independently of task periods (Ti). A periodic set of n tasks can be scheduled with the EDF algorithm as follows (1):

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{1}$$

The downside of the interrupt scheduling system in most current microcontroller applications is that it allows a large jitter in scheduling high-priority tasks. However, the possibility of executing out-of-core interrupts relieves the processor of additional overhead, thus eliminating several unnecessarily used context switches, including certain interrupt-specific clock cycles. The time required to change task contexts is the most significant factor in any RTOS. In the case of real-time systems, another workload factor is the time period required for the processor to execute the interrupt handling routine. If Q is the system tick and σ is the WCET corresponding to the periodic task, the overhead introduced can be calculated as the utilization factor Ut obtained by Equation (2).

$$U_t = \frac{\sigma}{Q} \tag{2}$$

For some RTSs, the preemptive CPU scheduler can be disabled for certain time cycles during the execution of the interrupt service routine (ISR). To achieve maximum performance, i.e., an IPC close to 1.0, it is necessary to modify the instruction and data memory handshake. The real-time event handling module provides the infrastructure that the applications need to dynamically monitor the task execution time, handle interrupts and count unallocated CPU cycles. This module can also be used for debugging and monitoring the timing behavior of each sCPUi, thus improving the performance of the hardware RTOS and offering low interrupt latency.

4. Preemptive Real-Time Scheduler Architecture and FPGA Implementation

This section describes the architecture of the hardware scheduler and its internal structure (see Figure 3). The real-time event handling unit is a scalable module based on the Mealy finite-state machine (FSM), which can be successfully used even in real-time applications. The contributions of this work are the result of theoretical and practical research in real-time scheduling. In this context, extra attention was paid to minimizing the overhead due to the operating system, allocating the time to the software scheduler and context switching, reducing the overall jitter effect. The proposed processor concept described in this paper is based on the five-stage pipelined MIPS processor [21,22] proposed in [23,24]. For implementing the new COP2 instructions, we used the MIPS32 instruction set. Possible sCPUi events are as follows: timer (TEvi), watchdog timer (WDEvi), deadlines (D1Evi and D2Evi), interrupts (IntEvi), mutexes (MutexEvi), inter-task communication events (SynEvi) and self-sustaining execution for the current sCPUi (lr_run_sCPUi). The abovementioned events can be validated with local registers (lr) en_Ti, en_WDi, en_D1i, en_D2i, en_Inti, en_Mutexi and en_Syni signals, the only exception being run_sCPUi. These signals must be stored in a special register named the task register (crTRi). The current nHSE scheduler is based on priorities. In addition to the watchdog timer registers (mrWDEVi), deadline 1 and deadline 2 (mrD1EVi, mrD2EVi), the effective monitoring registers (mrCntRun, mrCntSleepi, mr0CntSleep) are implemented in the hardware at

the level of each sCPUi. sCPU0 can access monitoring registers and can implement other scheduling algorithms via software. The sCPUiEvi signal, which is used to signal the occurrence of an expected event, is enabled by the stop_CPUi signal. The scheduler register-transfer level (RTL) equations are the following ("∧" AND logic, "∨" OR logic, "/" NOT logic, "CLK" HW_nMPRA_RTOS processor clock, "↑" positive edge trigger):

$$\mathrm{sCPU_{Ev_i}} \leftarrow \mathrm{mr_{stop_{CPUi}}} \wedge \mathrm{sCPU\_Ev_i} \tag{3}$$

$$\begin{aligned}\mathrm{sCPU\_Ev_i} \leftarrow\ & (\mathrm{lr_{en_{Ti}}} \wedge \mathrm{T_{Evi}}) \vee (\mathrm{lr_{en_{WDi}}} \wedge \mathrm{WD_{Evi}}) \vee (\mathrm{lr_{en_{D1i}}} \wedge \mathrm{D1_{Evi}}) \vee (\mathrm{lr_{en_{D2i}}} \wedge \mathrm{D2_{Evi}}) \\ & \vee (\mathrm{lr_{en_{Inti}}} \wedge \mathrm{Int_{Evi}}) \vee (\mathrm{lr_{en_{Mutexi}}} \wedge \mathrm{Mutex_{Evi}}) \vee \left(\mathrm{lr_{en_{Syni}}} \wedge \mathrm{Syn_{Evi}}\right)\end{aligned} \tag{4}$$

$$\mathrm{sCPUi_{\_ready}} \leftarrow /\mathrm{sCPU_{Ev_i}} \wedge /\mathrm{sCPU_{Ev_{i-1}}} \wedge /\mathrm{sCPU_{Ev_{i-2}}} \wedge \ldots \wedge /\mathrm{sCPU_{Ev_1}} \wedge /\mathrm{sCPU_{Ev_0}} \tag{5}$$

$$\mathrm{sCPUi_{rdy}} \leftarrow \mathrm{CLK} \uparrow \mathrm{sCPUi_{\_ready}} \tag{6}$$

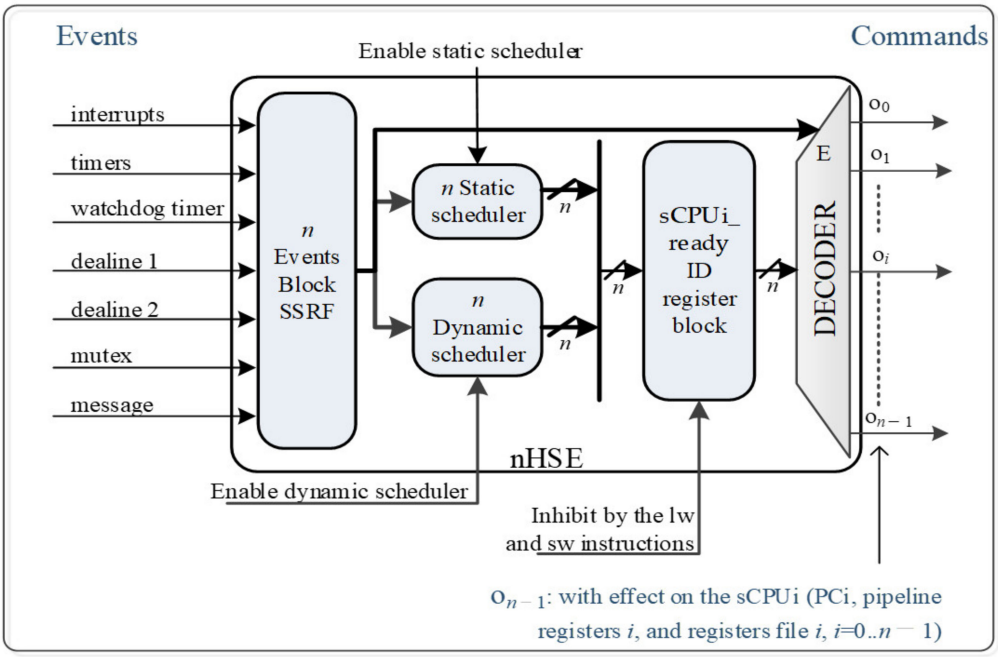


Figure 3. Architecture of the real-time hardware scheduler HW_nMPRA_RTOS.

The FSM outputs (Oi) are dependent on the scheduled sCPUi IDs and also on the current inputs represented by the events in Figure 3.

The block diagram contains the sCPUi_ready functional blocks and the register that stores the ID of the highest priority sCPUi (see Figure 3). Subsequently, the AND gate and the D flip-flop are activated when there is no other active sCPUi. The Figure 3 block shows the ID register of the active sCPU together with the synchronization logic, the static scheduler, the dynamic scheduler and the block related to the events. The en_CPU signal can be used mainly for power saving. The activation or deactivation of any sCPUi specific resources can be accomplished with O0 (en_pipe_sCPU0) through O*n*-1 (en_pipe_sCPUn-1) signals. The proposed schematic can be used for static scheduling if each task runs on a sCPUi. In this case, the static priorities are identified by the IDs of the tasks. In this context, interrupts borrow the priority and the behavior of the task. Thus, interrupt behavior is much more predictable in the context of a real-time application (a task can only be interrupted by interrupts attached to a task with a higher priority).

5. Validation Details Based on Priority Scheduling Model

Since most RTOSs used in the automotive industry use the stack to store function contexts, it is possible that while the application is running, the shared memory space is accidentally corrupted and the saved contexts of important functions become unusable.

The proposed processor concept with replicated resources provides a special solution to this problem because the tasks have individual contexts that are managed in hardware by a versatile scheduler without any additional overhead that leads to penalties over the scheduling algorithm and execution times of the application. The HW_nMPRA_RTOS architecture is an innovative one with very low response times to external stimuli. Improving these times, as well as minimizing the time spent on switching the task contexts, is also one of the main research purposes of this work. Thus, the hardware-accelerated RTOS uses a real-time scheduler that is part of the processor, with its control being performed through dedicated instructions that are sent over the pipeline assembly line.

5.1. The Implementation of Synchronization and Communication Mechanisms

When a task activates an event, it prepares the values specific to the SSR (signaling and synchronization register) of the SSRF (signal and synchronization register file) and executes an instruction specific for setting an event without indicating the address of the SSRF event. Each sCPUi has a hardware block in nHSE, which is used to generate SynEvi signals every time a free event becomes active (Figure 4a). With the help of the lr_en_Evi0, . . . , lr_en_Evis − 1 signals, each sCPUi can decide which event is taken into consideration. These signals are stored in the SSR local registers. There can be one or several SSRi registers, depending on the number of events implemented in the nHSE. The logic scheme in Figure 4b generates the address of the first available signal (which is 0 L) starting from the 0 address or eventually indicates that all events are active (on 1 L). In the case of the event activation instruction, the in_rdev_rd signal is 0 L and the output of the multiplexer takes over the output of the D-type flip-flops in the scheme. If Signal_0 (the value stored in SSR0) is 0 L, the signal denoted/Signal_0 is 1 L and writing will be performed in the flip-flop corresponding to Signal_0. Signal_0 on 0 L blocks the writing in all other flip-flops. The 1 L value generated by/Signal_0 is stored in the flip-flop, its output passing through the multiplexer and it will activate the three-state gate. The gate will provide the 0 L value at the input of the DEMUX circuit and, as a consequence, the 0 address will be activated. If Signal_0 is 1 L, 0 L is written in the first flip-flop that, after passing the demultiplexer, inhibits the three-state circuit and implicitly the 0 address. Going to the second flip-flop, if Signal_0 is 1 L, it validates the analysis for Signal_1. The analysis is similar to that for Signal_0; only Address_1 will be provided if Signal_1 is 0 L. The analysis can continue for the other events up to $s - 1$. The OR gate from the last flip-flop notifies when all events are active.

The gr_en_mem_full signal will be generated and retrieved by the activation instruction of an event. It can eventually be read through a global register (gr) available to all sCPUi or by 1-bit in the crEVi event register of every sCPUi. Figure 5a shows the generic signals (in_wrev_wr, Address_i and/gr_ev_mem_full) for writing in the flip-flop corresponding to the *i* signal and, respectively, in the other bits (bit_ij) of the SSRi register. The reading resets the flip-flop corresponding to the *i* signal by activating the in_wrev_wr, Address_i and hit signals and enables the reading of the content of the other bit_ij bits from the SSRi register. This way, the three-state gate is validated. This mode of activating the signals eliminates the time required to search for an available signal. The SSRF registers can be accessed from the level of any sCPUi because they are a resource shared by all semiprocessors. At the level of each sCPUi, there is a scheme similar to the one shown in Figure 5a that enables the generation of a SynEvi event every time an expected event is available. It can be decided at the level of each sCPUi which signal is taken into consideration with the help of the lr_en_Sii0, . . . , lr_en_Siis − 1 signals. These signals are stored in the local registers, named enable signal and synchronization events registers (ESSRs). There may be one or more ESSRi registers, depending on the number of events implemented in the SSRF. The D-type flip-flop that retrieves the information on the rising edge of the processor clock is used for synchronizing with the CPU clock. When a task *i* is executed by the sCPUi, the task is awakened by an event and the hardware scheduler stores the signal that generates the event. Again, searching in the SSRF registers could take an unacceptably long CPU

time. To avoid this situation, when executing a reading instruction for the corresponding SSRi register from the SSRF, the search is performed based on the CAM principle, as shown in Figure 5b. The hardware search starts with the zero address and ends with the first address for which there is a match between the ID of the destination sCPUi and the ID of the current sCPUi. The content of the SSRF address is assumed by the reading instruction that can determine if a match has been found and who issued the event, and eventually what message has been sent.

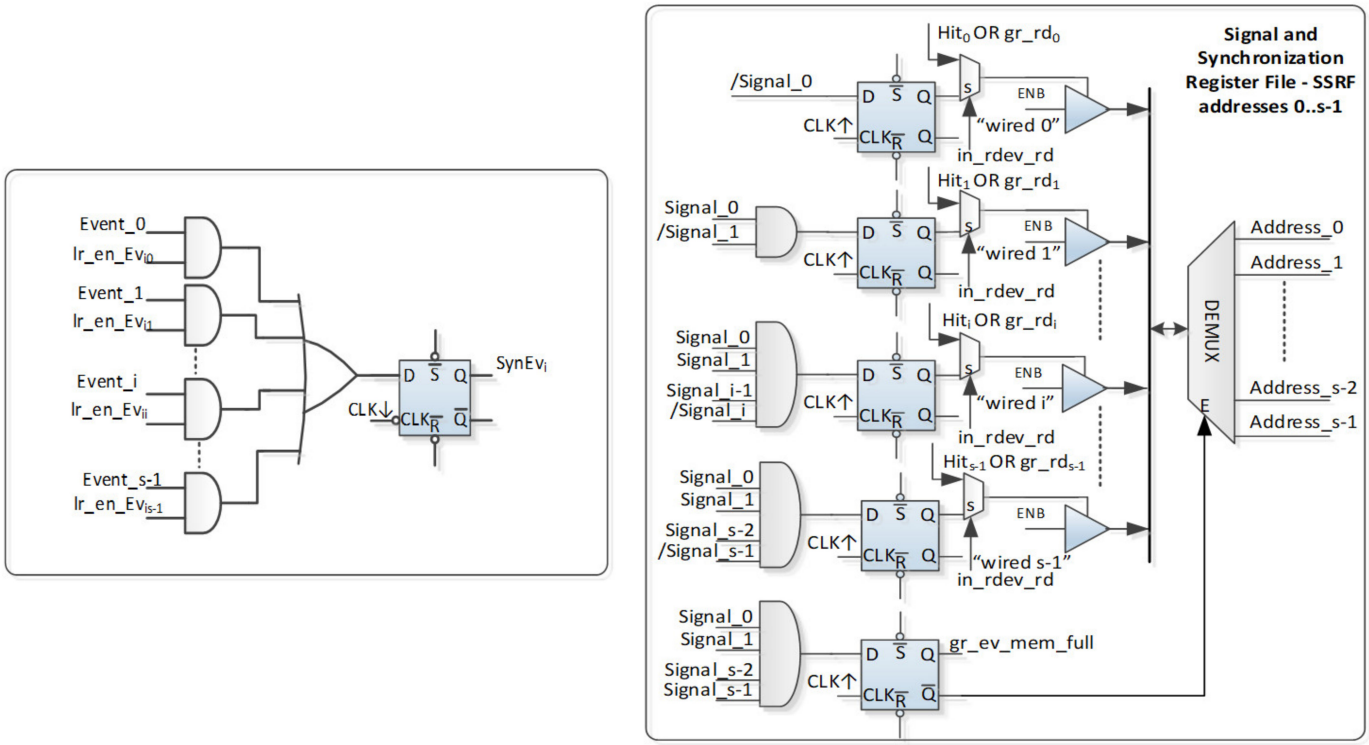


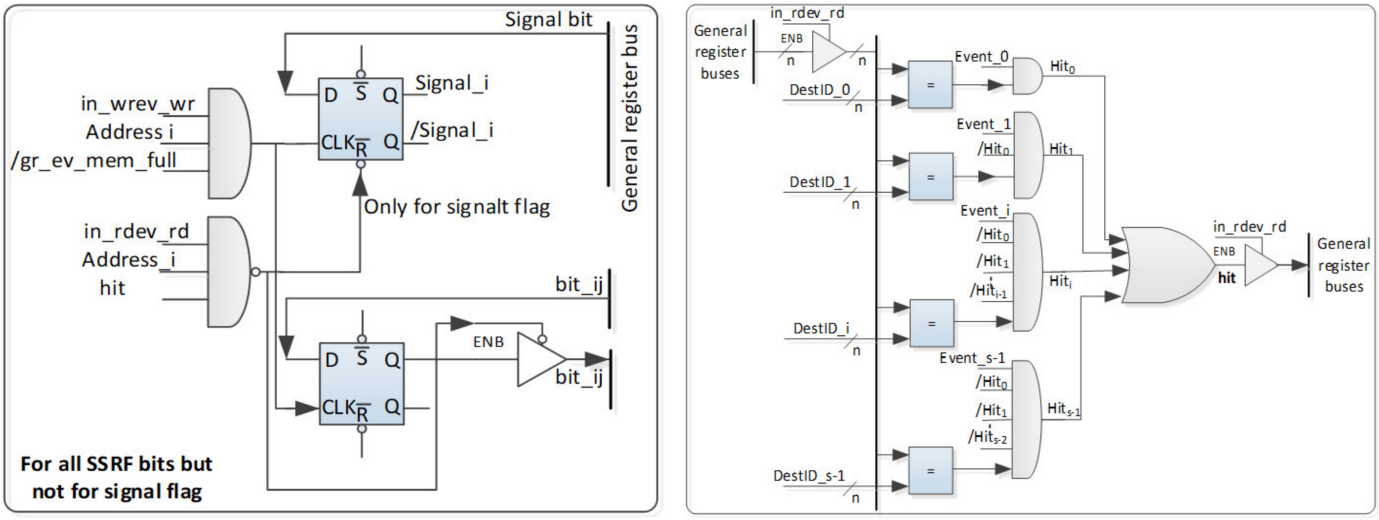
Figure 4. (**a**) nHSE event validation logic; (**b**) automatic address generator for the first free event based on nHSE.

The ID of the executed task and the DestID_0, . . . , DestID_s − 1 destination values are provided at the input of the comparison blocks. If there is a match and the validated event is active, the scheme generates a hit signal on the general register bus.

As shown in Figure 4b, the in_rdev_rd signal makes the multiplexer assume the Hiti OR gr_rdi-type input (reading as a global register). For working with events, the event Ri register instructions (Ri contains the source task ID, the destination task ID and the *k* bits of the message) can be used. In return, Ri contains the gr_en_mem_full that, before the execution of the instruction, was in the lowest bit position. If this bit is 1 L, the event activation failed. The writing operation of the grSSRi register in the SSRF consists of the following actions:

- The register used by the writing instruction must be loaded with a signal value (=1 L), the ID of the source task that must match the sCPUi ID, the destination task ID (=that of the sCPUj) that must be different from the source task ID, and the message value (not important for the hardware);
- The source task ID is used as an identifier (who sent the message) for the destination task;
- A search must be performed for the first valid register of the SSRF, meaning the register with signal bit on 1 L. The hardware scheme already has a valid address if there is one that can be used for identifying the grSSRi register in SSRF;

- If there is an available location, the swap is performed with the value that already exists in the SSRF register selected by the hardware. Otherwise, the swap operation cannot be performed;
- If the instruction performing the writing returns a value with signals on 1 L in the GPR, then the writing failed (the value from the used register remains unchanged, so the swap has not been performed). If the signal bit is 0 L, the writing has been completed successfully, meaning that the swap was completed.



(**a**) (**b**)

Figure 5. nHSE scheduler operation for real-time event handling: (**a**) read and write operations to SSRF registers; (**b**) content-addressable memory (CAM) read method of SSRi register and activation of the in_rdev_rd signal.

5.2. Validation of Mutex Events Handled by HW_nMPRA_RTOS

The COP2 instructions will be taken into consideration for the HW_nMPRA_RTOS implementation using the MIPS32 architecture. Figure 6 and Table 1 illustrate the practical measurements of the mutex event latency based on the scheduler implemented in HW_nMPRA_RTOS. In this test, the mutex event was validated by means of the lr_enMutexi bit in the crTRi register. The wait Rj instruction can synchronize the thread with seven events (time, deadline 1, 2, WDT, mutex, message and interrupt event), which allows implementation using a single instruction to obtain time-type functions and to gain access to critical resources by automatically acquiring a mutex (grMutex0, . . . , grMutexm − 1). Throughout the execution of the work instructions with the grMutexi[0] mutex, the nHSE_inhibit_CC internal signal is enabled to prevent this instruction from being interrupted. As can be seen in Figure 6, cursors C1, C2 and C3 indicate the context switches performed under the command of the nHSE_Task_Select and nHSE_EN_sCPUi signals. The waveform features obtained with the Vivado 2018.2 Design Suite by Xilinx, Inc. (San Jose, CA, USA) correspond to the logic implementation of HW_nMPRA_RTOS at the level of RTL based on the Verilog HDL synthetized implementation. The result of executing the 0x48c1ffff instruction at the moment indicated by marker C3 is the context switch between sCPU0 and sCPU3 performed at time moment T4. As can be seen, the time needed for switching contexts is no more than one clock cycle because the proposed processor architecture is based on the principle of remapping multiplied contexts, thus improving the RTS performance. The time moment marked by cursor C1 in Figure 6 indicates a context switch between sCPU3 and sCPU0 (nHSE_Task_Select[3:0] = 0x3, =0x0) since the sCPU0 has a higher priority scheduled to handle a time event. The grMutexi[0] = 0x80000003 value indicates that sCPU3 has acquired one of the four validated mutexes. At time moment T1, sCPU0 attempts to acquire the grMutexi[0] mutex, but although this semiprocessor has the highest priority, the

mutex cannot be assigned to it. The crTRi = 0x00000071 hexadecimal value indicates that sCPU0 expects mutex events, message events, interrupts and time events. These bits can be changed by the wait Rj instruction, the Rj register containing both the validated/inhibited events and the mutex ID expected by the sCPUi. Before executing the wait Rj instruction, it is possible to read the status bit in order to check if the mutex is available. Therefore, the execution of the wait Rj instruction enables the validation of multiple priority events through the crEPRi control register.

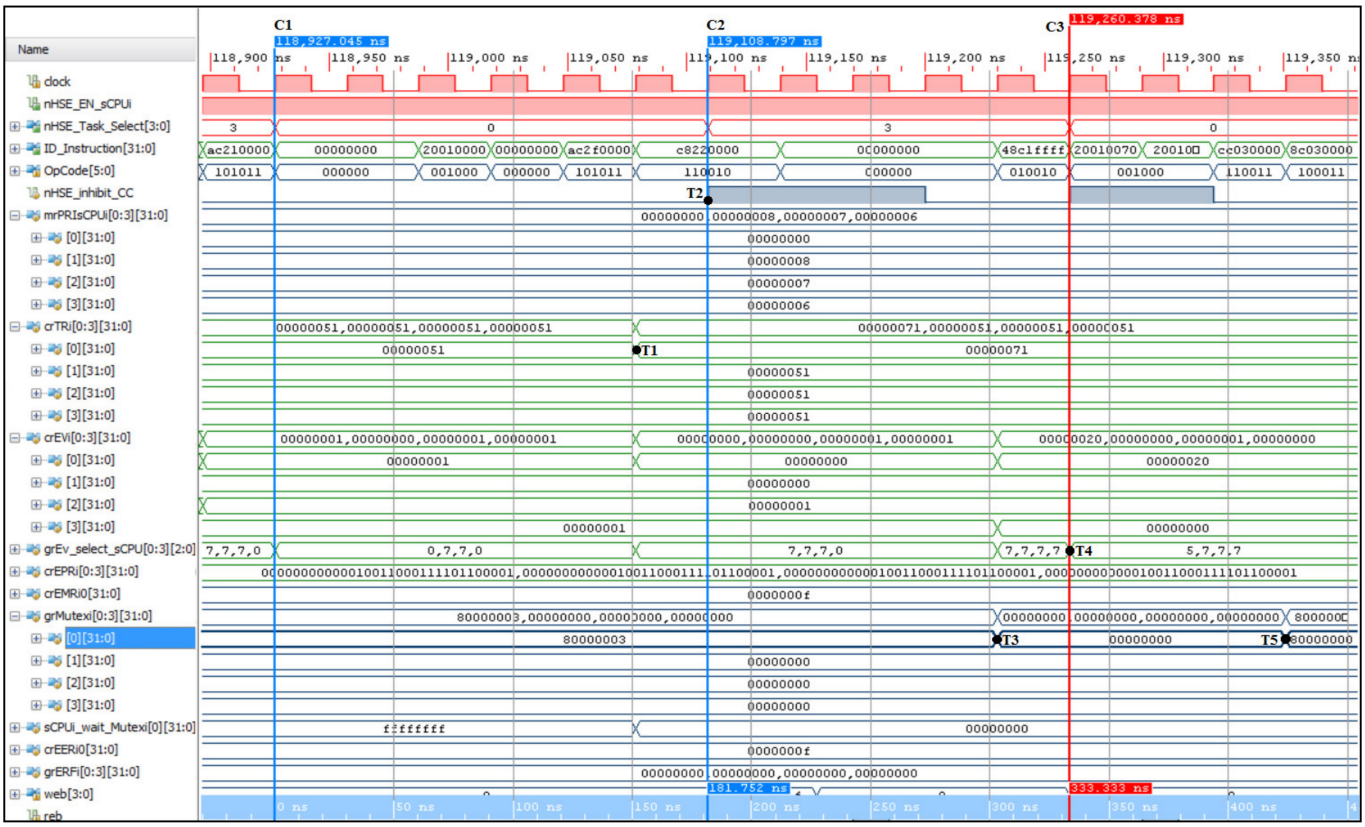


Figure 6. Signals corresponding to the mutexes obtained through the Vivado 2018.2 Design Suite by Xilinx.

At the time moment marked by marker C2 (see Figure 6), the context switching between sCPU0 and sCPU3 is performed, because sCPU0 waits for the release of the grMutexi[0] mutex, which is gained by sCPU3. Being a priority-based preemptive scheduler, the real-time event handling module will introduce in execution the sCPUi with the highest priority. In this case, the processor is associated with sCPU3, executing the instruction 0xC8220000 (see Table 1). The time moment T2 (or marker C2) indicates the nHSE_inhibit_CC signal that is set for the mutex access instruction to be indivisible. At time moment T3, grMutexi[0] is released, taking a 0x00000000 null value, the value in the crEVi = 0x00000020 control register indicating the occurrence of the mutex MutexEvi event corresponding to sCPU0. Marker C3 indicates the context switch between sCPU3 and sCPU0, and T4 denotes the content of the gr_EV_select_sCPU[0] = 0x5 register. This value indicates that sCPU0 handles a mutex-type event, sCPU0 acquiring grMutexi[0] at time moment T5. The priority-based preemptive scheduler accomplishes the search for the mutex event expected by sCPU0, as well as the context switch that lasts one clock cycle. It can be seen that the nHSE_inhibit_CC signal is set in order to inhibit a possible context switching of the preemptive scheduler, sCPU0, acquiring mutex grMutexi[0] at time moment T5 when grMutexi[0] = 0x80000000. Figure 6 illustrates the effect of executing the 0xC8220000 instruction, validating the implementation of the synchronization mechanism.

By executing the instructions given in Table 1, the ID_Instruction (Figure 6) provides the active instruction on the instruction decode (ID) pipeline stage corresponding to the sCPUi ($i = 0, \ldots, n-1$) selected for execution. At time moment T1, the instruction is extracted from memory and at T3 and T5, we can see the grMutexi[0] global register value in relation to the sCPU3 and sCPU0 instruction execution.

Table 1. The application sequence to validate a mutex event generated by the release of mutex grMutexi[0] (sCPU0 and sCPU3).

ID_Instruction [31:0] Signals (Machine Code)	MIPS32 Instructions (Included COP2)	Application Description
		sCPU3 execution
ac210000h	sw	store word MIPS instruction, save r1 in data memory
		Context switch: sCPU3 to sCPU0 (sCPU0 handling a event)
00000000h	nop	No operation
20010000h	addi	Add Immediate MIPS instruction, SignExtImm = 0000
00000000h	nop	No operation
ac2f0000h	sw	save gpr15 in data memory
c8220000h	ldgr	load word in grMutexi[0] gr
		Context switch: sCPU0 to sCPU3
c8220000h	ldgr	load word in grMutexi[0] gr
00000000h	nop	No operation
48c1ffffh	movcr	The wait Rj instruction causes the next context switch
		Context switch: sCPU3 to sCPU0
20010070h	addi	SignExtImm = 0070 h
20010071h	addi	SignExtImm = 0071 h
cc030000h	stgr	save grMutex[0] gr
8c030000h	lw	load word in gpr3 from data memory
		sCPU0 continues execution

Each sCPUi can have different priorities (crEPRi[3:0]) for time events, interrupts, mutexes and synchronization events through messages. This example was meant to test and validate the implementation of the hardware synchronization mechanism at the level of the priority-based preemptive scheduler. The hardware implementation of the communication mechanism and the mutex search based on the CAM principle is high-performance for the hardware RTOS implementation used for real-time applications [25]. As can be seen in Figure 6, the time elapsed from time moment T3, at which grMutexi[0] mutex is released, until sCPU0 is scheduled for execution is one machine cycle. Therefore, the access instructions for the special registers of the synchronization mechanism implement partially or totally the OSSemPend, OSSemPost and OSSemAccept functions of the real-time kernel µC/OS-II [26].

5.3. Resource Usage and Synthesis Results

In the implementation and validation of the processor described in this paper, the negative effects produced by the software RTOS overhead have been minimized, improving the context switch time and real-time determinism along with the processor performance. Table 2 shows the memory requirements for three possible implementations with 4, 8 and 16 sCPUi, including the hardware support for handling external interrupts. Therefore, resource multiplication for sCPU16 totals 4.299 kB of memory. However, the RAM required to save the contexts of nested function contexts is not added [27]. The command, control and status registers with a direct or indirect effect on the hardware scheduler are presented and described in the real-time scheduler specifications. These registers are also defined in the implementation of the event handling module, using the Verilog hardware description

language and the Vivado 2018.2 Design Suite. Since synchronization and inter-task communication mechanisms are achieved through shared memory areas, task context switching must ensure consistency of the data used for these mechanisms. Thus, data corruption in case the scheduler executes a preemptive algorithm is eliminated.

Table 2. Memory requirements for the datapath resource multiplication including real-time event handling unit.

CPU Configuration/Resource Required	Memory Required for Scheduler, Including Inter-Task Synchronization and Communication Mechanisms	Memory Required for PC, GPR and Pipeline Registers	Memory Required (Total)
nMPRA4/grMutexi4/grERFi4	0.262 kB	0.862 kB	1.124 kB
nMPRA8/grMutexi8/grERFi8	0.459 kB	1.724 kB	2.183 kB
nMPRA16/grMutexi16/grERFi16	0.851 kB	3.448 kB	4.299 kB

Regarding the operating frequency of FPGA circuits, due to their structure having a limitation in operation compared to ASIC circuits, tests were performed at 33 MHz. The SoC HW_RTOS_nMPRA project can be synthesized and mapped to another FPGA because an IP clocking wizard is used to generate the CPU clock signal.

Figure 7a presents the FPGA resource requirements for three versions of the proposed processor, with 4, 8 and 16 sCPUi/mutex/message events. The design of multi-tasking applications that highlight the power consumption and performance of this real-time hardware-based microprocessor (nMPRA + nHSE), as well as providing support for debugging, will also be taken into consideration. Figure 7b illustrates the power consumed by the FPGA circuit following the FPGA post-implementation of the HW_RTOS_nMPRA project, which includes the proposed processor with 4 sCPUis. By making a comparison between the dynamic scheduler and the static scheduler presented in [28], we can state that the implementation here involves additional consumption due to the dynamic scheduling of the real-time event handling module.

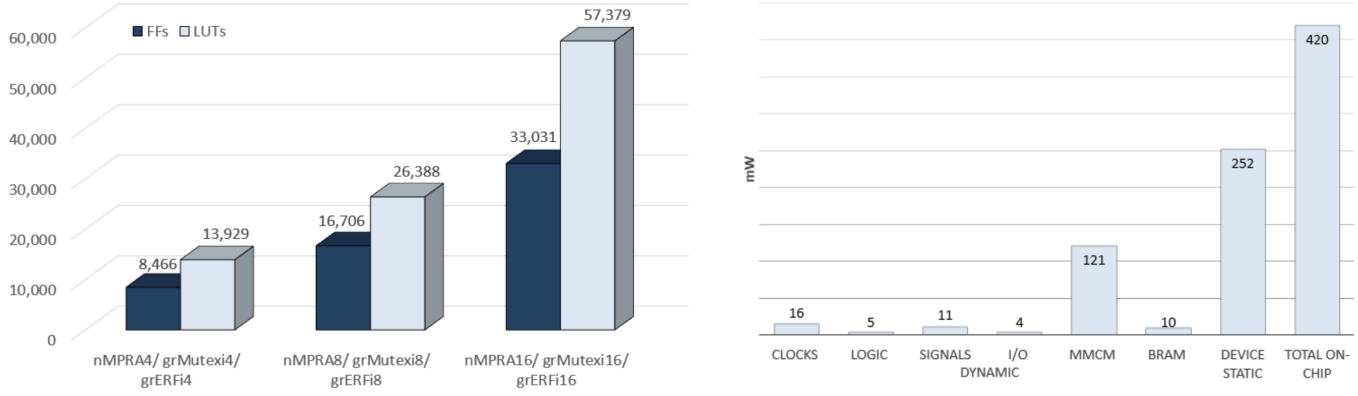
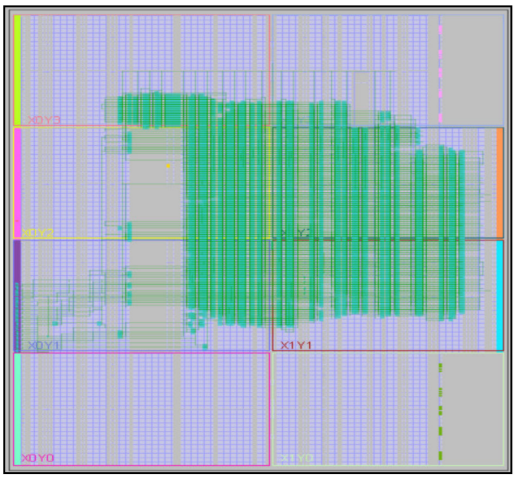


Figure 7. (**a**) FPGA resource requirements in terms of different HW_RTOS_nMPRA configurations; (**b**) power consumed by the HW_RTOS_nMPRA implementation, including support for the dynamic scheduler.

Figure 8 shows the distribution of logic cells and post-implementation FPGA resource requirements used to implement the proposed processor with four semiprocessors, the preemptive dynamic scheduler and four external interrupts, including the synchronization and communication mechanisms implemented in the hardware.



(**a**)

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	14906	63400	23.51
LUTRAM	814	19000	4.28
FF	8567	126800	6.76
BRAM	1	135	0.74
DSP	4	240	1.67
IO	31	210	14.76
BUFG	15	32	46.88
MMCM	1	6	16.67

(**b**)

Figure 8. (**a**) Distribution of the HW_RTOS_nMPRA (4sCPUi) project logic components, including the preemptive scheduler on the Nexys 4 DDR; (**b**) post-implementation FPGA resource requirements for nMPRA + nHSE (Artix-7 FPGA chip XC7A100T-1CSG324C).

6. Discussion

The HW_RTOS_nMPRA dynamic preemptive scheduler is responsible for deciding which task to select for the RUN state, thereby making the appropriate context change; attaching interrupts; counting the clock cycles used by each sCPUi separately; counting unused clock cycles; managing the two time limits for the tasks; the transition of tasks from the RECEIVE state to the READY state at the expiration of the time periods set for each. Thus, the scheduler selects for execution the highest priority sCPUi from those in the READY state. In other words, the task chosen for execution is the highest priority task based on priorities (mrPRIsCPUi). RECEIVE or PREEMPTED tasks are not enabled for execution.

The resource requirements for implementing the proposed hardware scheduler based on real-time event processing and the low power consumption make it ideal for Internet of Things applications requiring flexible processing of data streams generated by multiple sensors, thus providing scalable, flexible solutions at the highest quality standards. Even RTLinux [29], a commercial RTOS, has a 32 µs jitter for the scheduling operation (worst-case jitter for a Compaq iPAQ PDA based on a 200 MHz StrongArm).

Figure 9 illustrates the jitter for verifying real-time scheduler performance implemented at the COP2-MIPS32-level. In this test, an Analog Discovery 2 oscilloscope by Digilent (Henley Ct. Suite 3, Pullman, WA, USA) was used for measuring the jitter of the preemptive scheduler in handling external asynchronous interrupts generated from the Virtex-7 kit. The address of the LED [7] peripheral device is mapped in the address space of the data memory, and the state of this device is switched by extracting, decoding and executing the 0xadcc0000 MIPS sw instruction. Following the performed practical measurements, a response time of only 602 ns was obtained. To test the total response time and the jitter introduced by the scheduler, a system composed of 4 tasks was considered, with task 1 running on sCPU1 that would release a mutex to the task with the highest priority running on sCPU0 (sCPU2 and sCPU3 threat time-related events).

Figure 10 shows the tests performed to measure the kernel latency in the case of an event assigned to sCPU0, which is the highest priority event according to crEPRi (the register for prioritizing events at the sCPUi level, corresponding to crEVi). Figure 10a illustrates the test performed for the practical measurement of the kernel latency corresponding to the nHSE scheduler (58.2 ns), i.e., the change in the output of the FSM states that generate the next transition through the nHSE_FSM_state[7:0] signals (time moment T2 from Figure 6). Thus, tests were run to confirm that the hardware scheduler has a jitter of 1 clock cycle

plus the time needed to trigger the IntEvi event (signal ExtIntEv[0] external interrupt), but any of the events specified in Figure 3 can be triggered. In addition, Figure 10b shows the oscilloscope capture for measuring the time of the thread context switch in 1 clock cycle, where the second cursor measures the transition of the signal nHSE_Task_Select (Figure 6). The practical implementation of HW_RTOS_nMPRA in the FPGA validates the simulation presented in Figure 6, so the kernel latency for handling an IntEvi-type event is only 88 ns (the trigger time of the external signal ExtIntEv[0] plus the 2 clock cycles needed for the hardware scheduler and the thread context switch).

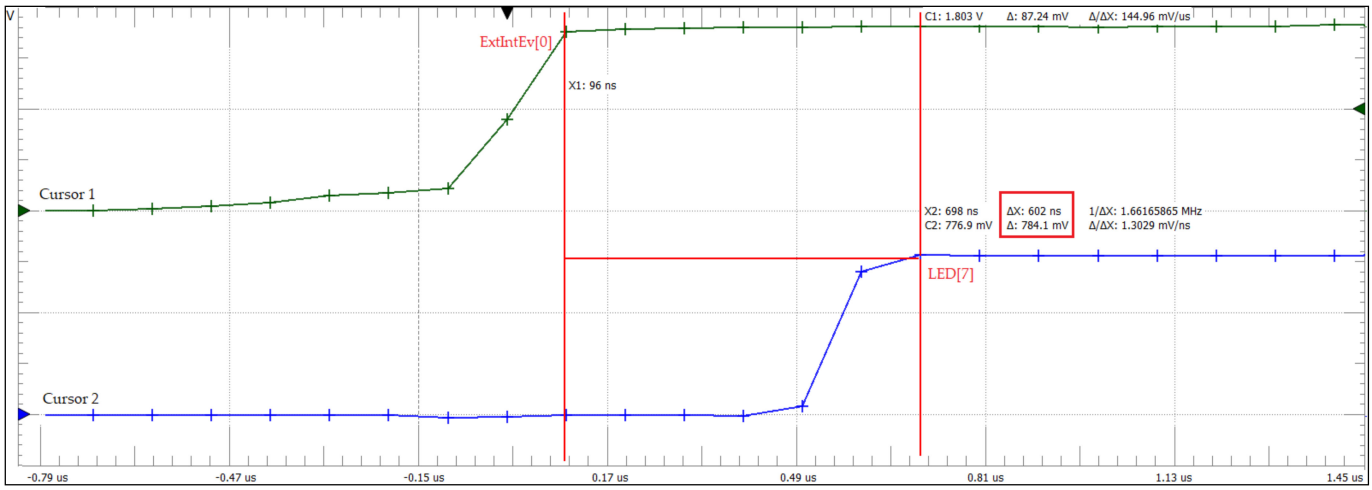
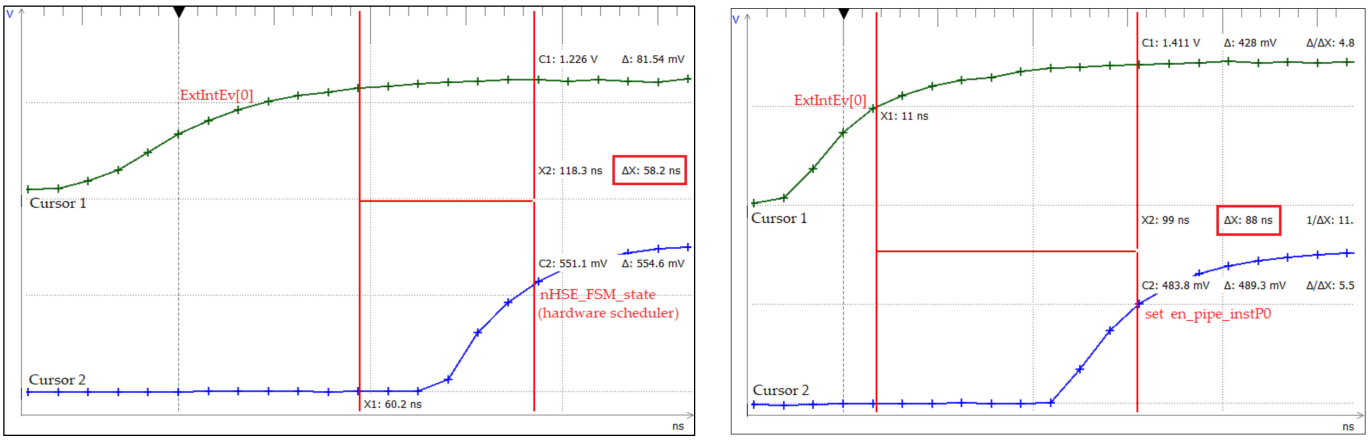


Figure 9. Kernel latency based on the hardware search in grMutexi[i] registers (602 ns, including the external interrupt threading).



(**a**) (**b**)

Figure 10. Kernel latency following project implementation in an FPGA: (**a**) changing the FSM state (58.2 ns); (**b**) context switch to sCPU0 (88 ns).

7. Conclusions

The proposed low-power programmable architecture can be used successfully in many critical applications, even in systems with multiple criticality specifications. As the main contribution, the HW_nMPRA_RTOS concept offers a response speed demonstrated both by tests and waveforms corresponding to the Vivado simulation and by practical operation obtained from outside the FPGA with an oscilloscope. An application sequence to validate a mutex event was created and initially tested on the simulator, where a time of one clock cycle was obtained for changing the thread contexts. With the FSM implemented in nHSE and real tests in FPGA after the mapping process, the oscilloscope connected to the pins of the FPGA was used to measure 88 ns as the response time of the hardware scheduler. The

602 ns period includes the execution time for all the sCPU0-sCPU3 instructions presented in Table 1. Compared to a software OS, a time period of a few microseconds is required to perform these functions using a microcontroller and a high-performance software RTOS. The novelty consists of the introduction to the hardware of support for mutexes and examples related to these tests.

Some of the hardware-accelerated RTOS characteristics are the following: guarantee for minimum latencies at interrupts; solidity and stability at the execution of tasks in the real time mode under the influence of interrupt overloads; a memory management unit that guarantees the best, scalable, deterministic use for the process-thread execution of industrial applications; preemptive execution; reduced memory imprint; compact implementation; high performance; strict control over the interrupt behavior; reducing the overhead of the RTOS's basic functions (scheduling, context switching and calls of the operating system functions); flexibility and minimum jitter in relation to event threats. In the future, the scheduler scheme implemented in the nHSE may include a dynamic scheduler from the EDF family. In this sense, nHSE allows the setting of a dynamic priority at the level of each sCPUi through a priority register (mrPRIsCPUi).

8. Patents

The central processing unit with pipeline registers is patented in Germany, Munich (DE202012104250U1, June 2012).

Author Contributions: Conceptualization, V.G.G. and I.Z.; software, I.Z. and V.G.G.; data curation, I.Z. and V.G.G.; writing—original draft preparation, V.G.G. and I.Z.; writing—review and editing, I.Z. and V.G.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the project "119722/Centru pentru transferul de cunoștințe către întreprinderi din domeniul ICT—CENTRIC—Contract subsidiar 15567/01.09.2020/DIW-PADCU/Fragar Trading", contract no. 5/AXA 1/1.2.3/G/13.06.2018, cod SMIS 2014 + 119722 (ID P_40_305), using the infrastructure from the project "Integrated Center for research, development and innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for fabrication and control", contract no. 671/09.04.2015, Sectoral Operational Program for Increase of the Economic Competitiveness co-funded from the European Regional Development Fund.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data sharing not applicable.

Acknowledgments: The authors would like to thank the editor and the anonymous reviewers for reviewing our manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Reichenbach, M.; Knodtel, J.; Rachuj, S.; Fey, D. RISC-V3: A RISC-V Compatible CPU with a Data Path Based on Redundant Number Systems. *IEEE Access* **2021**, *9*, 43684–43700. [CrossRef]
2. Wang, W.; Zhang, X.; Hao, Q.; Zhang, Z.; Xu, B.; Dong, H.; Xia, T.; Wang, X. Hardware-Enhanced Protection for the Runtime Data Security in Embedded Systems. *Electronics* **2019**, *8*, 52. [CrossRef]
3. Dörflinger, A.; Albers, M.; Kleinbeck, B.; Guan, Y.; Michalik, H.; Klink, R.; Blochwitz, C.; Nechi, A.; Berekovic, M. A comparative survey of open-source application-class RISC-V processor implementations. In Proceedings of the 18th ACM International Conference on Computing Frontiers, Ischia, Italy, 11–13 May 2021. [CrossRef]
4. Włostowski, T.; Serrano, J. Developing Distributed Hard-Real Time Software Systems Using FPGAs and Soft Cores. In Proceedings of the ICALEPCS 2015, Hardware Technology, Melbourne, Australia, 17–23 October 2015; pp. 1073–1078, ISBN 978-3-95450-148-9.
5. HW-RTOS. Available online: https://www.renesas.com/eu/en/software-tool/hw-rtos#overview (accessed on 29 December 2019).
6. Gaitan, V.G.; Cristina, G.N.; Ungurean, I. CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2014**, *23*, 1661–1674. [CrossRef]
7. May, D. The XMOS Architecture and XS1 Chips. *IEEE Micro* **2012**, *32*, 28–37. [CrossRef]

8. Kluge, F.; Wolf, J. *System-Level Software for a Multi-Core MERASA Processor*; Institute of Computer Science, University of Augsburg: Augsburg, Germany, 2009.
9. Ungerer, T.; Cazorla, F.; Sainrat, P.; Bernat, G.; Petrov, Z.; Rochange, C.; Quinones, E.; Gerdes, M.; Paolieri, M.; Wolf, J.; et al. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro* **2010**, *30*, 66–75. [CrossRef]
10. Clemente, J.A.; Resano, J.; Gonzalez, C.; Mozos, D. A Hardware Implementation of a Run-Time Scheduler for Reconfigurable Systems. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2010**, *19*, 1263–1276. [CrossRef]
11. Vermeulen, F.; Catthoor, F.; Nachtergaele, L.; Verkest, D.; De Man, H. Power-Efficient flexible processor architecture for embedded applications. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2003**, *11*, 376–385. [CrossRef]
12. Gschwind, M.; Salapura, V.; Maurer, D. FPGA prototyping of a RISC processor core for embedded applications. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2001**, *9*, 241–250. [CrossRef]
13. Xilinx. MicroBlaze Soft Processor Core. 2017. Available online: https://www.xilinx.com/products/design-tools/microblaze.html (accessed on 22 February 2019).
14. Amber Open Source Project: Amber 2 Core Specification. Available online: https://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-core.pdf (accessed on 22 February 2019).
15. Amber Open Source Project User Guide. Available online: https://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-user-guide.pdf (accessed on 22 February 2019).
16. Paul, R.; Shukla, S. Partitioned security processor architecture on FPGA platform. *IET Comput. Digit. Tech.* **2018**, *12*, 216–226. [CrossRef]
17. Zimmer, M.; Broman, D.; Shaver, C.; Lee, E.A. FlexPRET: A processor platform for mixed-criticality systems. In Proceedings of the 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Berlin, Germany, 15–17 April 2014; pp. 101–110. [CrossRef]
18. Oliveira, A.S.R.; Almeida, L.; Ferrari, A.B.; Almeida, A. The ARPA-MT Embedded SMT Processor and Its RTOS Hardware Accelerator. *IEEE Trans. Ind. Electron.* **2011**, *59*, 890–904. [CrossRef]
19. Tanase, C.A. An approach of MPRA technique over ARM cache architecture. In Proceedings of the 2016 International Conference on Development and Application Systems (DAS), Suceava, Romania, 19–21 May 2016; pp. 86–90. [CrossRef]
20. Dodiu, E.; Gaitan, V.G. Central Processing Unit with Combined into a Bank Pipeline Registers. DE Grant DE202012104250U1, 25 February 2013.
21. Patterson, D.A.; Hennessy, J.L. *Computer Organization and Design, Revised Fourth Edition: The Hardware-Software Interface*, 4th ed.; Elsevier: Amsterdam, The Netherlands, 2011; pp. 330–379. ISBN 978-0-12-374750-1.
22. MIPS®Architecture for Programmers Volume I-A: Introduction to the MIPS32®Architecture. Revision 3.02, Mar. 2011. Available online: https://courses.engr.illinois.edu/cs426/Resources/MIPS32INT-AFP-03.02.pdf (accessed on 22 February 2019).
23. Ayers, G. eXtensible Utah Multicore (XUM) Project at the University of Utah, 2011–2012. Available online: http://opencores.org/project,mips32r1 (accessed on 22 February 2019).
24. Meakin, B. Multicore System Design with Xum: The Extensible Utah Multicore Project. Master's Thesis, The University of Utah, Salt Lake City, UT, USA, 2010.
25. Zagan, I.; Găitan, V.G. Hardware RTOS: Custom Scheduler Implementation Based on Multiple Pipeline Registers and MIPS32 Architecture. *Electronics* **2019**, *8*, 211. [CrossRef]
26. Labrosse, J.J. *µC/OS-II The Real Time Kernel*, 2nd ed.; Taylor & Francis Group: Oxfordshire, UK, 2002; ISBN 1-57820-103-9.
27. Xilinx. VC707 Evaluation Board for the Virtex-7 FPGA User Guide. 2016. Available online: https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf (accessed on 22 February 2019).
28. Ciobanu, E.-E. The Events Priority in the nMPRA and Consumption of Resources Analysis on the FPGA. *Adv. Electr. Comput. Eng.* **2018**, *18*, 137–144. [CrossRef]
29. Yodaiken, V.; Dougan, C.; Barabanov, M. *RTLinux/RTCore Dual-Kernel Real-Time Operating System*; Technical Paper; FSMLabs Inc.: Bee Cave, TX, USA, 2003.