Hardware Real-time Event Management with Support of RISC-V Architecture for FPGA-Based Reconfigurable Embedded Systems

Ionel ZAGAN[1,2], Cristian Andy TĂNASE[1,2], Vasile Gheorghiță GĂITAN[1,2]
[1]*Stefan cel Mare University of Suceava, 720229, 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, Suceava, Romania*
zagan@eed.usv.ro

***Abstract*—Task context switching, unitary management of events, synchronization and communication mechanisms are significant problems for each real-time operating system. For real-time systems, another overhead factor is the processor's time to execute the routine of treating external asynchronous interrupts. The main objective of this paper is to describe, implement, and validate the preemptive scheduler module as part of the hardware accelerated real-time operating system, using the RISC-V instruction set and Verilog HDL. The new architecture contains the hardware structure used for static and dynamic scheduling of the tasks, real-time management of the events, and also defines a method used to attach interrupts to tasks. In order to accomplish this objective, it was necessary to structure CPU modules so as to ensure easy adaptation to other implementations (MIPS coprocessor, ARM or RISC-V).**

***Index Terms*—pipeline processing, field programmable gate arrays, architecture, operating systems, scheduling.**

I. INTRODUCTION

Nowadays, embedded systems are an ideal platform to implement projects in terms of the requirements of Internet of Things and Industry 4.0. Various real-time time systems (RTOS), used in the automotive domain, robotics or industrial automation, must guarantee a response within specified time constraints. For these systems, the time required to switch task contexts and the RTOS jitter introduced in treating aperiodic external events are very important parameters. In a central processing unit (CPU), the context refers to the data in the registers and program counter at a specific time moment. In reality, safety-critical systems use intensive RTOS that implements complex mechanisms for isolating the critical components from uncritical ones. The aim of moving the operating system, or some of its components in hardware, is to reduce the non-determinism sources introduced by asynchronous events. For this reason, a solution is needed to provide predictability and accurate response from the system when its state differs from the normal functioning mode, such as service mode. However, a RTOS has a well-specified maximum deadline for each task that it executes to support applications with precise timing needs.

This work is supported by the project ANTREPRENORDOC, in the framework of Human Resources Development Operational Programme 2014-2020, financed from the European Social Fund under the contract number 36355/23.05.2019 HRD OP /380/6/13 – SMIS Code: 123847.

To design and implement a real-time event management, this paper proposes the nMPRA concept (Multi Pipeline Register Architecture - *n* degree of multiplication) based on the RISC-V architecture. The main characteristic of the hardware accelerated processor is defined by the context switch operation that is made in a single clock cycle, with a worst case scenario of three clock cycles for special instructions used in case of external memory accesses [1]. The implementation of the real-time event handling module provides superior performance in terms of response time and reduced time needed to switch task contexts; the architecture is suitable for real-time small-scale applications due to the resource consumption needed to multiply the multiplexed storage elements. The proposed processor is a deterministic hardware implementation, due to the integrated preemptive scheduler and ZScale – RISC-V architecture. In this project, the three-stage pipeline VScale implementation with the Verilog VScale version proposed as an ”open source” at Berkeley University, will be used in order to design the nMPRA and the nHSE (Hardware Scheduler Engine for *n* tasks). Moreover, RISC-V instruction set and the Virtex-7 VC707 development kit will be used for synthesis and FPGA implementation. This experimental project aims to implement, test and validate a dynamic scheduler module as part of the proposed microprocessor [2], [3], using the RISC-V instruction set, as well as the implementation of the Z-Scale three-stage pipeline architecture. Implementation will be validated in FPGA for various configurations, considering *n* - the number of tasks, *i* - the number of interrupts, *m* - the number of mutexes, and *s* - the number of communication events. The functions specific to RTOS, such as the tasks scheduler, the synchronization and inter-task communication mechanisms, play a special role in real-time systems (RTS). The lack of hardware implementation of these mechanisms is a challenge for current research in the field. Thus, FPGA circuits [4] can be used to design and test a microprocessor implemented in hardware in order to provide low response times, minimal jitter, and low power consumption [5], [6]. The FPGA implementation of hardware accelerated RTOS based on real-time event handling module is a deterministic concept, due to the integrated preemptive scheduler and ZScale - RISC-V architecture.

With respect to the novelty of the paper, we believe that

Digital Object Identifier 10.4316/AECE.2020.01009

the paper brings the following contributions:

- Based on the current technological developments, the proposed event handling module concept has been implemented using the RISC-V instruction set.
- Tests have been performed in order to validate the functionality of the proposed preemptive scheduler module used for unitary management of events and the Verilog VScale design.
- Obtaining the number of logical components used on the FPGA chip for a diverse range of configurations (4, 8 and 16 degrees of resource multiplication) and implicit real-time event handling module (number of interrupts, mutexes and message events) in correspondence with the RISC-V architecture.

This paper is structured as follows: the first section contains a brief introduction and Section II describes similar papers in the field of real-time embedded systems. Section III presents the concept and theory of nMPRA processor operations based on RISC-V architecture and Section IV describes the practical results obtained during the validation of the theoretical elements presented in the previous section. In Section V the authors present the paper final conclusions.

II. Related Work

This section analyzes and compares several CPU implementations and projects proposed in the field of real-time scheduling. Theoretical and practical aspects related to the schedulers of the analyzed architectures are highlighted, as well as the features of the real-time characteristic. The main features of the most representative CPU and RTOS research projects with functions implemented in hardware refer to the implementation of the scheduler [7], the pipeline assembly line, and the type of the implementation.

The processor core proposed in [8] is composed of two five stages of assembly pipelines. The first one is dedicated to a single hard real-time thread (HRT), and the second pipeline is dedicated to non-HRT (NHRT). In a quad-core version, each core is composed of four hardware slots. Thus, each core can simultaneously execute one HRT and three NHRT. To the HRT is assigned the highest priority, being isolated from the other NHRT from the core through the real-time scheduler. The threads priorities are fixed and Round-Robin is the chosen scheduling scheme.

Kuacharoen, Shalan and Mooney [9] mention that a commercial software scheduler can have a very high overhead. When the clock frequency is 100KHz, for just 64 tasks, this overhead represents almost 46% of the CPU usage. The time earned by making the scheduling operation in hardware can be used to execute one or more useful tasks in the system. The time required to switch contexts is an intrinsic boundary of the kernel [10], which does not depend on the scheduling algorithm or task set structure. If Q is the tick of the system and σ is the worst-case execution time (WCET) corresponding to the periodic task [11], the introduced overhead can be calculated as the U_t utilization factor obtained through the relation (1).

$$U_t = \sigma/Q \quad (1)$$

The effects of scheduling operation due to preemptions can be taken into consideration by adding the U_t to the total usage factor corresponding to the periodic task set. Typically, preemptive RTOS have priority inversion scenarios, when high priority tasks are suspended by asynchronous interrupts assigned to low priority tasks. Ordering tasks and interrupt events in the same address space has the role to eliminate this disadvantage. Although it eliminates over control due to scheduling operations, the coprocessor-processor architectures cannot eliminate the following types of over control:

- Bus arbitration overhead (for schedulers that communicate on the same bus to which other devices are also connected - eg, direct memory access (DMA)).
- Time of effective data transfer through the CPU bus.
- Interrupt processing times (for events where the main processor is triggered by the interrupt mechanism).
- Time to save and restore tasks contexts, operations that are still performed in the conventional processors mode.

In terms of commercial processor architectures that benefit from advanced context management mechanisms, we mention the TriCore 1.3.1 [12] from Infineon and Intel 80960 [13]. The TriCore architecture efficiently manages and maintains the tasks' contexts through hardware. Context switching occurs when an event or instruction causes a break in program execution, in this context the CPU need to resolve the event before continuing with the program. The TriCore architecture uses a number of 32 registers, of which 16 are used for addresses, and the other 16 are used for data. The register management mechanism can automatically save registers to the stack only partially. Of the 32 registers, only 16 are saved in Context Save Area (CSA). In order to save the other half of the registers, the user must initiate this operation manually. Although this architecture is optimized for efficient context switching, the main impediment is related to the time needed for contexts to save operation using the stack. Thus, a period of time is required to perform manual saving of the lower half of the general purposed register set, if it is desired to completely save the context of the executed task.

In the I960 processors family, the user benefits from a set of local registers that are automatically remapped over a memory area named Stack Frame, indicated by the Frame Pointer (FP). Register g15 is reserved for the current FP which contains the address of the first byte in the current (topmost) stack frame. The architecture has a circular frame buffer available for a limited number of tasks. Even if it is a relevant architecture, the I960 has two drawbacks. Context saves involve stack memory transactions, which is slower than the internal processor registers. For determinism needed by RTS, the architecture can be improved if for these memory frames there is no rigid correspondence of the task - stack frame.

III. Concept and Theory of Operation of Hardware Accelerated Processor based on RISC-V

The issues covered in this paper relate to the architecture used for the proposed implementation, the integration of hardware scheduler registers into the nMPRA concept [14], the resource replication, the assembly line (the number of stages), and the synchronization and communication mechanisms [15]. Also, the integration into the hardware of a preemptive scheduling algorithm based on priorities has

been taken into consideration. The main modules of the pipeline assembly line are:

- The Instruction Fetch/Instruction Decode - Execute (IF/IDEX) and the Instruction Decode - Execute/Memory - Write Back (IDEX/MEMWB) pipeline registers.
- The fetch stage with the Program Counter (PC) loader module and the PC for addressing the I$ instruction memory (as alternative, ALU, EPC and EVEC outputs from CSR may be used as an instruction). Moreover, the instruction memory and the simple adder (+4) can be used in order to increment the PC register.
- The instruction decode and execution stage includes the immediate value generator, the implementation of the register file module as well as the forwarding multiplexers (*fwd1* and *fwd2*), the *A* and *B* multiplexers, the ALU combinational unit, the data storage control unit, the CSR module multiplexer with values in the register file or with immediate values from instructions, conditional jump block, and the control unit.
- The stage for accessing the memory and writing back data in the register file involves the following modules: the data memory, the CSR block including the hardware real-time scheduler registers, the charging unit, the 4-bit adder for PC-IDEX/MEMWB and the multiplexor for selecting the value to be written back in the register file.

The project will be synthesized on xc7vx485tffg1761-2 FPGA chip, implementing nMPRA + nHSE concept for n = 1, 2, 4, 8, 16, 32, a variable number of interrupts (i = 1 ÷ 32) and mutexes (m = 1 ÷ 32), and a variable number of signals (s = 1 ÷ 32).

Throughout this paper, the word signal (s) is used to name all inter-task synchronization and communication events validated and treated by real-time scheduler. The nHSE + nMPRA blocks must be implemented in Verilog HDL to enable the easy synthesis of different configurations through *n*, *i*, *m* and *s* parameters. The decision to multiply resources will be taken by analyzing the correctness of the VScale implementation, with three pipeline stages (Fig. 1), and the RISC-V instruction set [16], [17]. The basic idea is that any storage element (except data and instruction memories) in the datapath must be multiplied by *n* and its output signal must be multiplexed *n* to 1. The design is meant to enable the testing of the proposed functionalities.

As shown in Fig. 1, Z-Scale is the architecture chosen for nMPRA whereas for the implementation the Vscale architecture has been chosen. This architecture was implemented and validated using the Virtex-7 FPGA and VC707 development kit produced by Xilinx.

The extremely fast context switch of the nMPRA concept is accomplished in hardware by remapping the active context of the task to be executed. Regarding the power consumption by the FPGA implementation, if all tasks are inactive, the pipeline corresponding to sCPUi's may be set in sleep mode using the hardware scheduler control registers. This architecture shown in Fig. 1 is designed with three pipeline stages, namely:

- The stage for extracting instructions (FETCH) and IF/IDEX pipeline registers.
- The stage for decoding and executing instructions followed by IDEX/MEMWB pipeline registers.
- The stage for memory access, CSR and write back to register file.

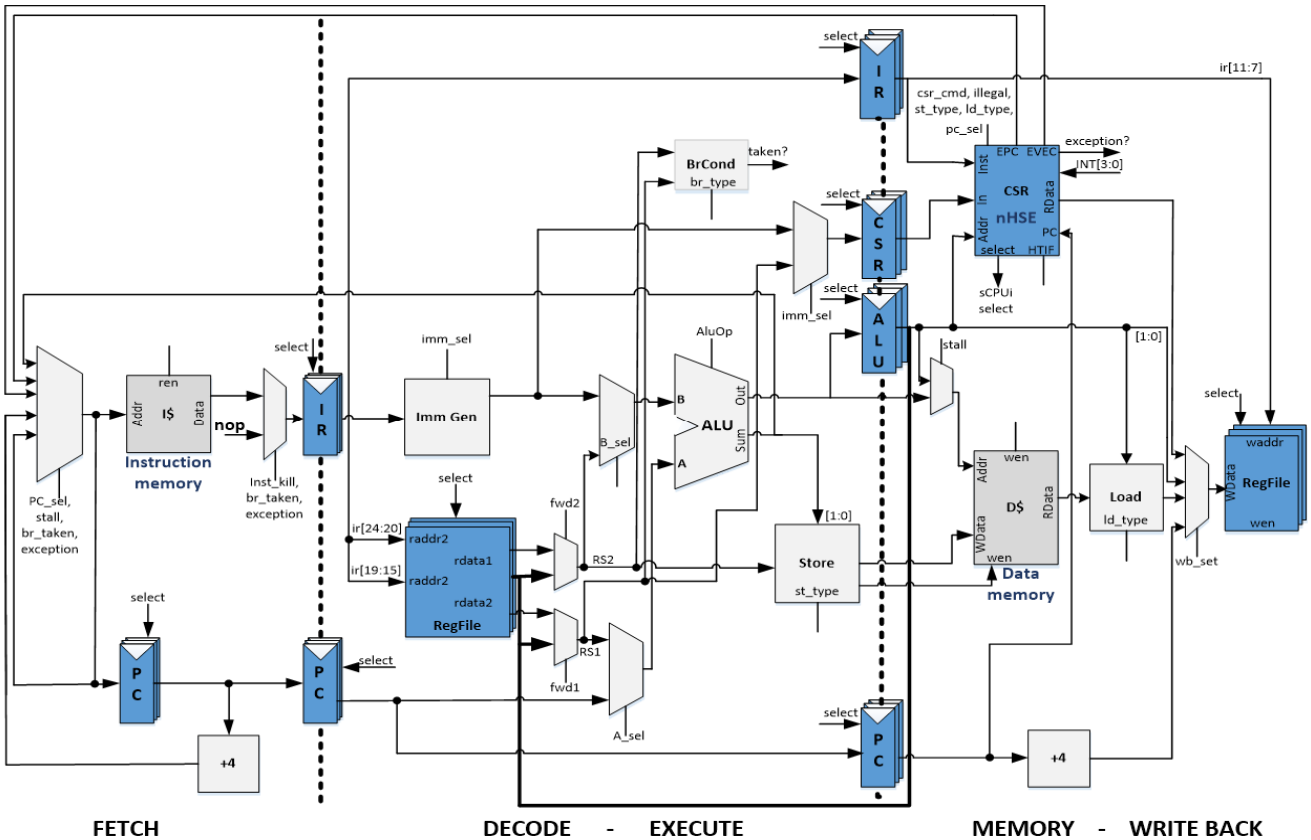

Figure 1. The RISC-V (ZScale) architecture based on resource multiplication and the integrated hardware scheduler; **PC**-Program Counter, **IR**-Instruction Register, **RegFile**-Register File, **ALU**-Arithmetic and Logic Unit, **CSR**-Control and Status Register, **EPC**-Exception PC, **EVEC**-Exception Handler Addr.

Subsequently are described some representative signals of the ZScale - nMPRA datapath implementation. The *PC_src_sel* (select PC source) signal is set by the control unit (Fig. 2), which has the role to select the next value with which the *PC_IF_REG* (NEW-PC) register is loaded. The *imm_type* signal selects the immediate value type in accordance with the instruction group. The implicit value for *alu_op* is *`ALU_OP_ADD*, so that these signals select the operation that needs to be executed by the ALU module, and the *add_or_sub* variable selects the addition or subtraction operation. Considering that *funct7[5]* is *inst_DX* (32 bit input in the control unit), the Verilog code for updating this value is the following:

assign add_or_sub = ((opcode == `RV32_OP) && (funct7[5])) ? `ALU_OP_SUB : `ALU_OP_ADD;.

The *dmem_en* signal indicates the access to the data memory (in reading or writing), and the *dmem_en_unkilled* signal identifies the two types of instructions working with the memory (LOAD and STORE). The *dmem_wen* (data memory write enable) signal sets a write operation in the data memory, while the output variable *dmem_size* defines the size of the data memory. The output variable *csr_imm_sel* (CSR immediate select) selects the immediate value for the CSR module directly from the instruction (= 0) or bypass data or from the register file (= 1). The output variable *wr_reg_WB* (write register in WB pipeline stage) indicates that the MEMWB stage will have a write back operation in the register file. The local register *dmem_en_WB* represents the MEMWB stage control signal for data memory validation (LOAD or STORE). In the instruction decode stage, the *wr_reg_DX* variable indicates that for the decoded instruction, a write back to the register file in the MEMWB stage may take place. The output variable *reg_to_wr_WB* stores the address of the register to be written in the MEMWB stage.

As can be seen in Table I, nMPRA implements mutexes in hardware and every *grMutexi* global register contains a bit for storing the state of the mutex and *m*-1 bits for the owner sCPUi identifier. The *Mutex Register File* (MRF) registers can be accessed from any sCPU and therefore they are shared resources for all sCPUi. Thus, each sCPUi generates a *MutexEvi* event (*crEVi[5]*) every time a blocked mutex is released (*Mutex i* bit from *grMutexi* global register). The block and release operations of a mutex are performed in a single processor cycle, as an atomic operation. The real-time event handling module uses a number of *grSRFi* global registers to compose the *Signals Register File* (SRF).

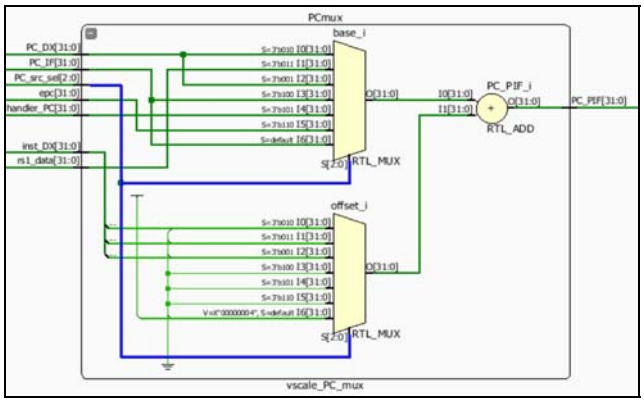

Figure 2. *PC_src_sel* signal effect in RTL representation after synthesis of the RISC-V (ZScale) architecture including the hardware scheduler

In order to implement the inter-task communication mechanism (Table II), each *grSRFi* (global register Signals Register File *i*) use one bit to store the event status (*Event i*), *2nj* bits for storing the tasks ID, their source (*s_IDnj-1÷s_ID0*) and destination (*d_IDnj-1÷d_ID0*), and *k* bits for storing the message (*Mess k-1÷Mess 0*).

The hardware block, implemented at the level of a preemptive scheduler, generates automatically the address (starting from 0) of the first free event and signals if all events are active (set to value 1). Since the Content Addressable Memory (CAM) search in the *grSRFi* registers is performed in hardware, the jump to the trap cell assigned to message events (the *crEVi[6]* bit named *SynEvi*) is done in only two clock cycles. An important aspect for hardware accelerated RTOS is the Verilog implementation of sCPUi timers (for example, as supervisor registers, etc.) and of the debug port in order to access internal data. Based on ZScale-nMPRA specifications were designed the Verilog HDL code for VScale resource multiplications, including extensions, and the Veriolg code for real-time event handling module.

The nMPRA implementation based on the RISC-V architecture uses also the following datapath signals, which are grouped according to the execution activity that they affect. The *stall_IF* signal preserves the Instruction Fetch stage data, this being directly influenced by the state of the *imem_wait*, redirect, *stall_DX* and exception signals. The wire output signal *kill_IF* is propagated in the *prev_killed_DX_reg* and *prev_killed_DX_reg* bistables, blocking the content of the *PC_IF_reg[31:0]* register. The CSR address convention uses the CSR address bits to encode the default access privileges.

TABLE I. THE IMPLEMENTATION OF THE MRF GLOBAL REGISTERS AND THE CORRESPONDING DATA AT A PARTICULAR EXECUTION TIME MOMENT

grMutex i	**31**	**30…5**	**4**	**3**	**2**	**1**	**0**
	Mutex i		Task ID bit4	Task ID bit3	Task ID bit2	Task ID bit1	Task IDbit0
grMutex 0	0/1 (Mutex 0)		0	0	0	0	1
grMutex 1	0/1 (Mutex 1)		0	0	0	1	0
..	..	..	..	..	..	..	..
grMutex m-1	0/1 (Mutex m-1)		0	0	1	1	1

TABLE II. THE STATE OF THE EVENT, THE TASK ID, SOURCE AND DESTINATION AND THE CORRESPONDING MESSAGE STORED IN THE SRF

Address	**Register**	**2nj + k + 1**	**nj - 1**	**.**	**0**	**nj - 1**	**.**	**0**	**k - 1**	**.**	**1**	**0**
		Event i	s_IDnj-1	.	s_ID0	d_IDnj-1	.	d_ID0	Mess k-1	.	Mess 1	Mess 0
Address 0	grSRF 0	0/1 (Event0)	0		0	0		1	0		1	1
Address 1	grSRF 1	0/1 (Event1)	0		0	0		0	0		0	0
…	…	…	…	…	…	…	…	…	…	…	…	…
Address e-1	grSRF e-1	0/1 (Evente-1)	0		0	0		1	0		0	1

This simplifies error checking in hardware and provides a larger CSR space, but constrains the mapping of CSR in the address space. The implementations could enable a high privilege level to gain access to other registers of the CSR, at a low privilege level, in order to enable the interception of these accesses. This change should be transparent for low privilege level software. The *wb_src_sel_WB* register with *WB_SRC_SEL_WIDTH* bit stores the selection for the source that will generate the new value when updating the register file in the WB stage. The *wb_src_sel_DX* variable selects the source in the Instruction Decode / Execute pipeline stage for the new value in the write backstage. The one bit signal *stall_WB* stops the assembly line, beginning with the WB stage, and the *dmem_wait* signal indicates that the data memory did not perform the read or write operation. Regarding the instructions and logic for branch operations, the *branch_taken* signal indicates that the branch operation takes place. Thus, the *branch_taken_unkilled* register indicates the decryption of the BRANCH instruction and the value *cmp_true* given by ALU. In this context, the wire input *cmp_true* validates that the condition for the BRANCH is true. The local variable *jal* implies the execution of a JAL instruction, while *jal_unkilled* sets the decoding of the same instruction. The *jalr* variable indicates execution of a *jalr* instruction, while the local variable *replay_IF* activates *kill_IF*, disables *ex_IF*, and determines that *PC_src_sel* = *`PC_REPLAY*. The local *illegal_instructon* variable indicates a non-existing instruction, while the *illegal_csr_access* input signal indicates an illegal access to the CSR. The *load_use* signal mark the presence of a hazard situation if a LOAD instruction is in the WB stage, and the *load_in_WB* signal validates a load operation in the WRITE_BACK stage.

IV. Practical Results Based on RISC-V Concept

The hardware accelerated processor architecture was implemented by multiplying all existing registers in the RISC-V (Z-scale) architecture shown in Fig. 1. All the memory elements were multiplied 32 times, so n = 32. Registers in the hardware real-time event handling block were mapped to the CSR area, between the following addresses: 0x200-0x2BF, 0xA00-0xAFF and 0xE00-0xEFF.

The initialization of internal scheduler registers is done by using the access instructions of these reserved areas. An assembler program code example that configures the real-time event handling unit internal registers is listed in Table III. Next, we will show the operation of the RISC-V processor when activates the timer events for sCPU0, sCPU1 and sCPU2 at different time intervals. The INT0, INT1, INT2, INT3 interrupts are individually assigned to the semi-processors sCPU7, sCPU5, sCPU7, sCPU4.

Prioritization of interrupts is given by the sCPUi to which it is assigned. Interrupts are disabled by the sCPU0 interrupt routine (sCPU0 = maximum priority, sCPU31 = minimum priority). Fig. 3 shows that at the occurrence of two simultaneously interrupts, the first one with the highest priority (sCPU5) is processed, *uiSelectCPU[4:0]* = 5. The delay caused by interrupt processing is maximum of two clock cycles.

Table III. The Application Sequence Used to Validate the Real-Time Hardware Scheduler

Application code description	Assembler instructions for real-time hardware scheduler validation
sCPU0 and sCPU2 treats a time event and sCPU7 handle INT0 external interrupt.	lui x16, F0000 //*x16 = F0000* ori x16, x16, FF //*x16 = F00000FF* csrrw 202, x16 //*cr0MSTOP = F00000FF* //*is activated sCPU0, 1, 2, 3, 4, 5, 6, 7, 28, 29, 30, 31* . . . . . lui x16, 0 //*x16 = 0* ori x16, x16, 7 //*x16 = 7* csrrw AB0, x16 //*grINT_ID0 = 00000007* //*INT0 is assigned to sCPU7* . . . . . . lui x16, 400 //*x16 = 0* ori x16, x16, F //*x16 = 0000000F* csrrw 281, x16 //*mrTEV2 = 0000000F* //*the timer recharge value for sCPU2* . . . . . . lui x16, 0 //*x16 = 0* ori x16, x16, 1 //*x16 = 1* csrrw 240, x16 //*crTR0 = 00000001* csrrw 209, x16 //*cr0PageReg = 1 change to page 1* //*enable timer interrupts for sCPU0 and enable registers page for sCPU1 (the register dedicated only sCPU1 is accessible).*

An interrupt can be attached to one task only [2], while a task can have attached more interrupts (even all of them). No constraints have been used, this being performed automatically at the FPGA implementation stage. After activating the real-time event handling unit, the registers implemented in hardware are accessed through the *wAddress* and *wRdData/wWrData* signals. The *bIdleCPU* signal is used when the entire system enters in sleeping mode. This situation occurs when all the sCPUi are in sleeping state. The scheduler exit from this state at the occurrence of an active interrupt validated and attached to a particular sCPUi. Fig. 4 shows that after the higher priority interrupt is deactivated, the lower priority interrupt is processed (*uiSelectCPU[4:0]* = 7) [18]. When INT3 (with the highest priority) is activated, sCPU4 (*uiSelectCPU[4:0]* = 4) is selected. It is also observed that as long as the interrupts are active, the events from the timers 0, 1 and 2 are not processed, the interrupt events having the highest priority.

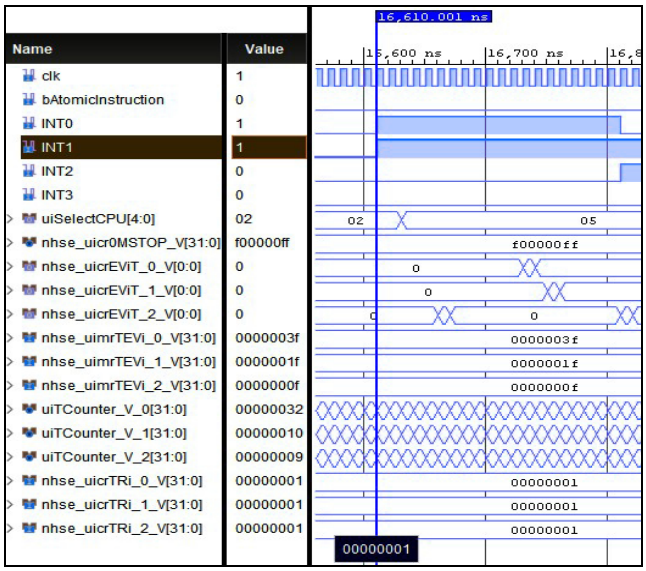

Figure 3. Simultaneous interrupts activations and threating based on prioritization scheme implemented in hardware

Fig. 5 shows the timer events handling when there are no interrupts. Timer events occur depending on the reload values defined by the following registers: *mrTEV0* = 0x3f, *mrTEV1* = 0x1f and *mrTEV2* = 0xf. Timer counters (*uiTCounter*) decrease until they reach 0 value. Then, they generate a timer event (*crEVi[0]* = 1), triggering the sCPUi for which the event was generated [19]. When the timer event is disabled (*crTR1[0]* = 0), the *uiSelectCPU[4:0]* signal is no longer 1 logic when the timer event occurs at sCPU1. The selection of these events can be achieved by executing a simple assembler instruction [2]. In order to be tested and validated in the FPGA, the RISC-V (Z Scale) architecture has been extended to develop software applications for the new hardware scheduler concept.

The attempts to access a non-existing CSR raise an exception to illegal instructions. Also, in case of a CSR access without an adequate level of privileges, or trying to write a read-only register, generates exceptions indicating an illegal instruction execution [19].

Each sCPUi runs the code that is between the start label and the *jrx29* instruction. For example, the code running on the sCPU0 is located between *start0:* and *jrx29*. The code running on sCPU3 is between *start3: lui x5, 0* //x5 = 0 and *jrx29 //go to start3*. The instruction code address of each sCPUi starts at address 200h for sCPU0, 300h for sCPU1, and ends at address 2100h for sCPU31. The response time of the nMPRA processor can be simulated and measured when an asynchronous external event occurs; also the time required to switch contexts can be determined. Thus, an assembly code has been implemented in order to program nHSE registers to generate different time events, but also to respond to external events, such as asynchronous interrupts [20], [21]. Activating the events at the level of each sCPUi can be achieved executing a simple nHSE instruction. The interrupts are very versatile and do not require a dedicated controller, the priority being the same as the sCPUi on which it is attached.

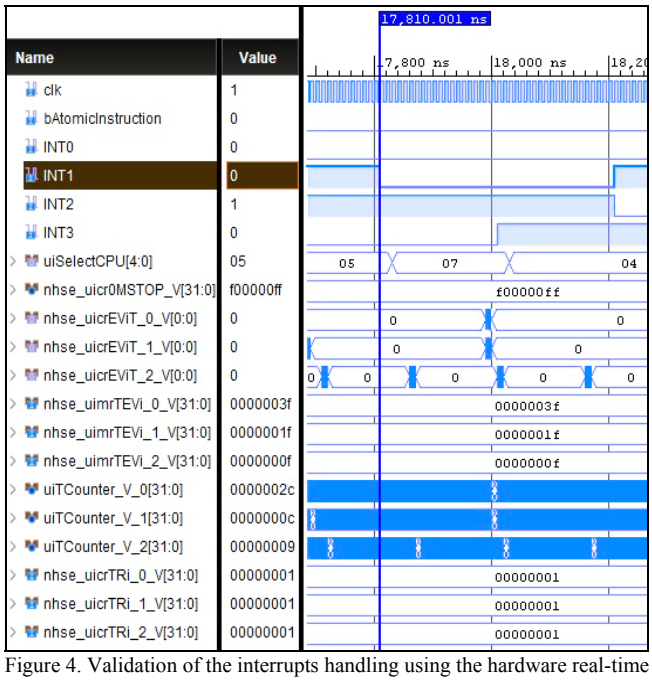

Figure 4. Validation of the interrupts handling using the hardware real-time event support

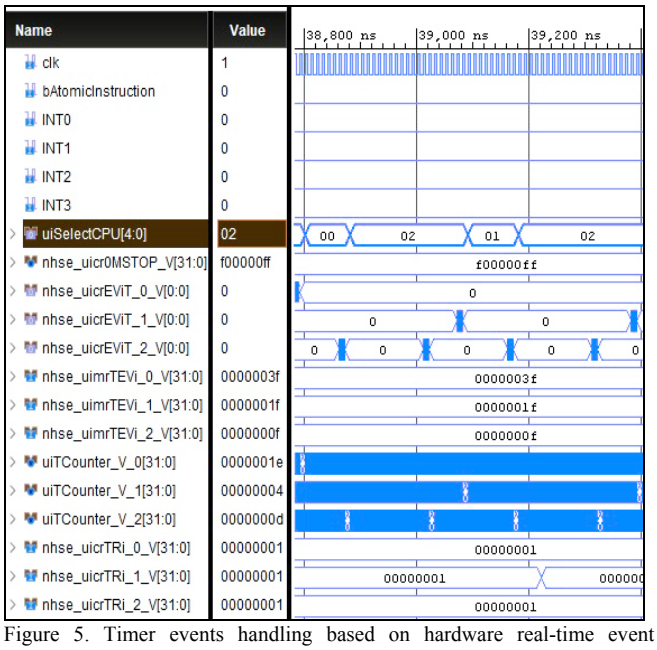

Figure 5. Timer events handling based on hardware real-time event management unit

In order to generate a delay, a loop has been implemented for each sCPUi separately. After the execution exit from this loop, the instruction *csrrw 780, x28* indicates through the Virtex-7 development kit LEDs the number assigned to the processor and jumps back to the start address stored in the *x29* register. Thus, when a sCPUi is active, its ID can be seen by turning on the combination of binary LEDs specific to the attached sCPUi.

The highest priority semi-processor, sCPU0, has multiple code lines because it has to configure the real-time event handling unit. Thus, it writes the *cr0MSTOP* register with the value 0xF00000FF, thus activating 12 sCPUi's.

sCPU0 loads registers *mrTEV0* = 0x0060003F, *mrTEV1* = 0x0050001F, *mrTEV2* = 0x0040000F, *mrTEV31* = 0x00300007 in order to generate timer events on semi-processors 0, 1, 2, and 31 at different time intervals; it also programs the *grINT_ID0* = 0x00000007, *grINT_ID1* = 00000005, *grINT_ID2* = 00000007, *grINT_ID3* = 00000004 registers to assign four existing interrupts to a particular sCPUi. It can be noted that interrupt 0 and interrupt 2 have assigned the same sCPU7 semi-processor, so the *grNrINT* register will store the address of the highest priority sCPUi. Timer events for the semi-processors 0, 1, 2 and 31: *crTR0* = 0x00000001, *crTR1* = 0x00000001, *crTR2* = 0x00000001 and *crTR31* = 0x00000001 are also enabled.

Regarding the response time-test for the proposed architecture, the highest priority task threat an external interrupt associated with the FPGA pin connected to the Virtex-7 test button. After executing the instructions sequence needed to initialize the hardware scheduler, the highest priority task waits for the events validated by the *crTRi* register (in this case, the external interrupt). When the associated signal on the Virtex-7 kit changes the state (time moment 1 on Fig. 6.a), the external interrupt is generated and the hardware scheduler will go into the execution mode of the task with the highest priority. This will set the second test signal illustrated through time moment 2.

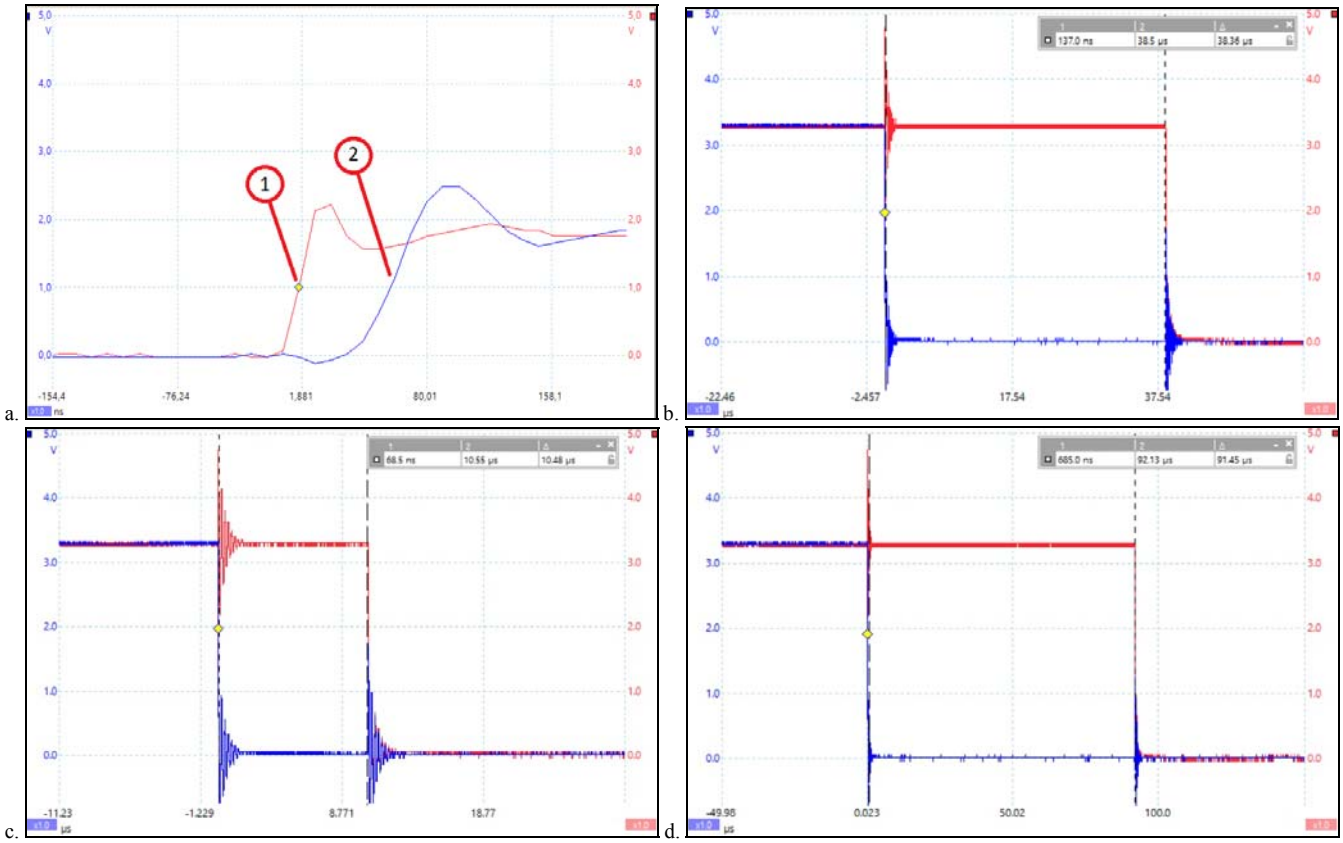

Figure 6. a) The context switching time of the most priority task in accordance with the logic implemented in the hardware scheduler at the CSR level; b) Experimental tests for event synchronization when considering µC/OS-II RTOS and STR912FAW44; c) External interrupts and event synchronization with RTX RTOS and STM32F429; d) Response time for event synchronization when using FreeRTOS and ARM Cortex-M4 microcontroller (STM32F429)

Fig. 6.a shows the test results for the hardware RTOS architecture based on a RISC-V processor at 33MHz. Time moment 1 represents the external event activation, and time moment 2 indicates when the preemptive scheduler has activated the task (sCPU0) with the highest priority (68ns). Thus, by multiplying each memory resource from the original pipelined datapath, is obtained the architecture illustrated in Fig. 1. The figure shows that each pipeline register is multiplied, and the context switch operation is done using the *uiSelectCPU (select)* signal. The register file has also been multiplied. The 5-bit *uiSelectCPU* signal is generated by the hardware scheduler implemented within the CSR unit [22]. Fig. 6.b shows the response time Dt = 38.36µs for event synchronization mechanism when using µC/OS-II real-time operating system and five-stage 32-bit RISC ARM966E-S microcontroller running at 25MHz. Fig. 6.c and Fig. 6.d shows the results obtained with RTX RTOS (Dt = 10.48µs) and FreeRTOS (Dt = 91.45µs) using STM32F429 microcontroller and event synchronization. It can be observed the response time from the moment when the synchronization event appears until the state of the microcontroller pin changes. In the context of external interrupts handling, the jitter depends on the current instruction executed by the CPU, Nested Vectored Interrupt Controller (NVIC), interrupt service routine (ISR) length, interrupt priority and Cortex-M4 hardware architecture (typical latency is 12 cycles followed by saving general purpose registers R0-R3, R12, Link Register, PC and Program Status Register).

Fig. 7 shows the distribution of logic cells used to implement the proposed processor with the preemptive dynamic scheduler (n = 32 sCPUs). The FPGA implementation incorporates the hardware handling block for time related events, external interrupts, and also synchronization and communication events. In this context, the HW-RTOS provides sCPUi management, mutexes, messages, hardware timers and asynchronous interrupts. During the implementation, tests were carried out for different configurations of the nMPRA architecture. The analysis refers to the percentage of resources used in the FPGA circuit and the processing frequency for different degrees of multiplication n, i, m, s = 1, 2, 4, 8, 16 and 32. These results are presented in Table IV.

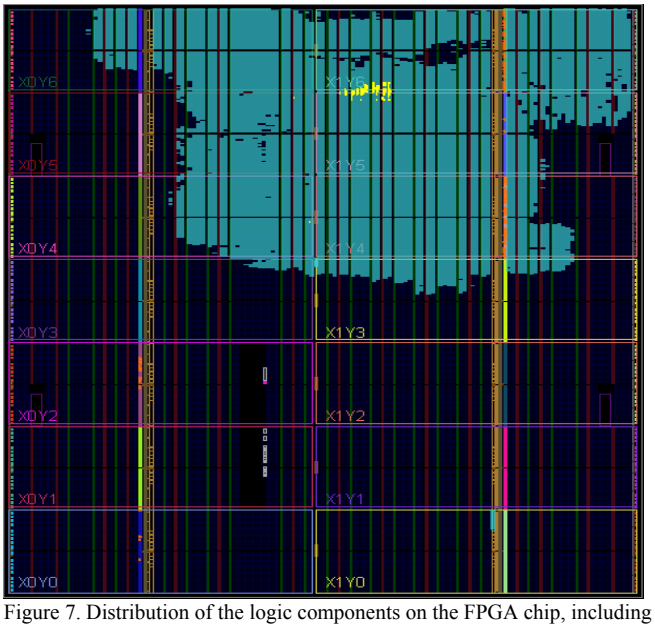

Figure 7. Distribution of the logic components on the FPGA chip, including the hardware handling block for events (n = 32 sCPUs)

TABLE IV. TESTS FOR DIFFERENT CONFIGURATIONS OF THE NMPRA ARCHITECTURE

Virtex-7 FPGA Resource (XC7VX485T-2ffg1761C)	**32 sCPU/i/m/s, 20 MHz**	**16 sCPU/i/m/s, 30 MHz**	**8 sCPU/i/m/s, 30 MHz**	**4 sCPU/i/m/s, 40 MHz**	**2 sCPU/i/m/s, 40 MHz**	**1 sCPU/i/m/s, 100 MHz**
LUT (Look Up Table)	10.32	5.37	3.45	2.70	2.35	1.14
LUTRAM	2.89	2.30	2.01	1.86	1.79	0.70
FF (Flip-Flop)	3.08	1.54	0.77	0.39	0.20	0.18
IO (Input/Output pins)	12.00	12.00	12.00	12.00	12.00	12.00
BUFG (Global Clock Buffer)	37.50	37.50	37.50	37.50	9.38	3.13
PLL (Phase-locked Loops)	7.14	7.14	7.14	7.14	7.14	7.14

The real-time aspects of task context switching time validation, FPGA implementation, and also the distribution of the logic components on the FPGA chip are presented in order to verify the theoretical aspects proposed through this paper.

V. CONCLUSION

The present hardware scheduler implementation is characterized by a fast response for events because the proposed architecture replaces the stack saving methods with a remapping algorithm that enables the execution of the new task starting with the next clock cycle. In this context, the task context switching is very fast, between 1 - 3 machine cycles (there are no search operations to find a free event). By performing practical tests, we can observe the effect of multiplied resources related to the FPGA chip. Also, a series of comparisons have been made with real-time microcontroller operating systems that are implemented in software, such as uC/OS-II, KeilRTX or FreeRTOS. This experimental project can be used to test practical applications developed for the hardware RTOS architecture, the next step is to implement this concept directly into an ASIC (application-specific integrated circuit).

Future research directions will continue the studies related to the hardware implementation of RTOS, taking into consideration the resource multiplication architecture and its hardware scheduler defining component.

ACKNOWLEDGMENT

This work is supported by the project ANTREPRENORDOC, in the framework of Human Resources Development Operational Programme 2014-2020, financed from the European Social Fund under the contract number 36355/23.05.2019 HRD OP /380/6/13 – SMIS Code: 123847.

REFERENCES

[1] I. Zagan, V. G. Găitan, "Hardware RTOS: Custom Scheduler Implementation Based on Multiple Pipeline Registers and MIPS32 Architecture," Electronics, vol 8, no. 2:211, 2019. doi:10.3390/electronics8020211

[2] V. G. Găitan, N. C. Găitan, I. Ungurean, "CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 9, pp. 1661-1674, Sept. 2015. doi:10.1109/TVLSI.2014.2346542

[3] E.-E. Ciobanu, "The Events Priority in the nMPRA and Consumption of Resources Analysis on the FPGA," Advances in Electrical and Computer Engineering, vol. 18, no. 1, pp. 137-144, 2018. doi:10.4316/AECE.2018.01017

[4] R. Paul, S. Shukla, "Partitioned security processor architecture on FPGA platform," IET Computers & Digital Techniques, vol. 12, no. 5, pp. 216-226, 2018. doi: 10.1049/iet-cdt.2017.0178

[5] W. Wang, X. Zhang, Q. Hao, Z. Zhang, B. Xu, H. Dong, T. Xia, X. Wang, "Hardware-Enhanced Protection for the Runtime Data Security in Embedded Systems," MDPI Electronics, vol. 8, no. 1:52, 2019. doi:10.3390/electronics8010052

[6] A. Melnyk, V. Melnyk, "Self-Configurable FPGA-Based Computer Systems," Advances in Electrical and Computer Engineering, vol. 13, no. 2, pp. 33-38, 2013. doi:10.4316/AECE.2013.02005

[7] S. Roman, H. Mecha, D. Mozos and J. Septien, "Constant complexity scheduling for hardware multitasking in two dimensional reconfigurable field-programmable gate arrays," IET Computers & Digital Techniques, vol. 2, no. 6, pp. 401-412, November 2008. doi: 10.1049/iet-cdt:20070060

[8] F. Kluge and J. Wolf, "System-Level Software for a Multi-Core MERASA Processor," Institute of Computer Science, University of Augsburg, Tech. Rep. 2009-17, October 2009.

[9] P. Kuacharoen, M. Shalan, V. J. Mooney III, "A Configurable Hardware Scheduler for Real-Time Systems," Proc. Engineering of Reconfigurable Systems and Algorithms, pp. 95-101, 2003. ISBN:193241505X

[10] J. Echague, I. Ripoll, A. Crespo, "Hard real-time preemptively scheduling with high context switch cost," Proceedings Seventh Euromicro Workshop on Real-Time Systems, Odense, Denmark, pp. 184-190, 1995. doi: 10.1109/EMWRTS.1995.514310

[11] S. Altmeyer, G. Gebhard, "WCET analysis for preemptive scheduling," 8th International Workshop on Worst-Case Execution Time WCET Analysis (WCET'08), Prague, Czech Republic, pp. 105-112, July 2008. doi: 10.4230/OASIcs.WCET.2008.1664

[12] TriCore 1, 32 bit Unified Processor Core, Volume 1, Core Architecture V 1.3 & V 1.3.1, Infineon Technologies AG, 81726 Munich, Germany, Jan. 2008, pp. 4-3 ÷ 4-13.

[13] Intel i960 Jx Microprocessor Developer's Manual. Intel Corporation, Order Number: 272483-002, Dec. 1997, pp. 7-1 ÷ 7-10.

[14] http://www.google.com/patents/DE202012104250U1?cl=en, Central processing unit with combined into a bank pipeline registers. DE 202012104250 U1. Owner Dodiu E., Găitan, V. G.

[15] C. A. Tănase, "An approach of MPRA technique over ARM cache architecture," in 2016 International Conference on Development and Application Systems (DAS), Suceava, Romania, pp. 86-90, 2016. doi: 10.1109/DAAS.2016.7492553

[16] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. Patterson, K. Asanovic, "The RISC-V instruction set," 2013 IEEE Hot Chips 25 Symposium (HCS), Stanford University, CA, USA, 2013. doi: 10.1109/HOTCHIPS.2013.7478332.

[17] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, "The RISC-V Instruction Set Manual Volume I: User-Level ISA, Version 2.1," Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016, pp. 9-23.

[18] I. Zagan, C. A. Tănase, V. G. Găitan, "FPGA IMPLEMENTATION OF A CUSTOMIZED PROCESSOR BASED ON RISC-V ARCHITECTURE – CONCEPT AND THEORY OF OPERATION," Proceedings of 148th IASTEM International Conference, Rome, Italy, 2018, pp. 24-29.

[19] Y. Lee et al., "An Agile Approach to Building RISC-V Microprocessors," IEEE Micro, vol. 36, no. 2, pp. 8-20, 2016. doi: 10.1109/MM.2016.11

[20] E. E Moisuc, A. B. Larionescu, V. G. Găitan, "Hardware Event Treating in nMPRA," in 12rt International Conference on Development and Application Systems – DAS, Suceava, Romania, pp. 66-69, 15–17 May, 2014. doi:10.1109/DAAS.2014.6842429

[21] S. Kelinman and J. Eykholt, "Interrupts as threads," ACM SIGOPS Operating Syst. Rev., vol. 29, no. 2, pp. 21–26, Apr. 1995. doi:10.1145/202213.202217

[22] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.7," Technical Report UCB/EECS-2015-49, EECS Department, University of California, Berkeley, May 2015, pp. 7-13.