Enhanced Interrupt Response Time in the nMPRA based on Embedded Real Time Microcontrollers

Nicoleta Cristina GAITAN[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), Suceava, 720229, Romania*
cristinag@eed.usv.ro

***Abstract*—In any real-time operating system, task switching and scheduling, interrupts, synchronization and communication between processes, represent major problems. The implementation of these mechanisms through software generates significant delays for many applications. The nMPRA (Multi Pipeline Register Architecture) architecture is designed for the implementation of real-time embedded microcontrollers. It supports the competitive execution of n tasks, enabling very fast switching between them, with a usual delay of one machine cycle and a maximum of 3 machine cycles, for the memory-related work instructions. This is because each task has its own PC (Program Counter), set of pipeline registers and a general registers file. The nMPRA is provided with an advanced distributed interrupt controller that implements the concept of "interrupts as threads". This allows the attachment of one or more interrupts to the same task. In this context, the original contribution of this article is to presents the solutions for improving the response time to interrupts when a task has attached a large number of interrupts. The proposed solutions enhance the original architecture for interrupts logic in order to transfer control, to the interrupt handler as soon as possible, and to create an interrupt prioritization at task level.**

***Index Terms*—architecture, operating systems, registers, scheduling, software.**

I. Background and Motivation

The most important features of the RTOS (Real Time Operating System) are predictability and response time guarantee to external or internal events. The use of RTOS implemented in software can generate higher response times to treat interrupts when they occur one immediately after the other. In this case, ISR (Interrupt Service Routine) are used, which must be implemented by the user. Due to the jitter generated by ISR routines, the response time can greatly increase, if the microcontroller does not allow nested interrupts. This could lead to missed deadlines. The problem occurs when a task waits the occurrence of different events (message, semaphore, mutex, etc.), because most RTOS allow waiting for a single event at a time (e.g. µC-OS/III, FreeRTOS, eCos, Keil RTX, and so on). In this case, events in loop must be expected, each one with a waiting timeout. Therefore, because of the jitter generated by waiting for the other events, the response time to certain events can greatly increase. The same problem occurs if a task waits for more than one interrupt. The task should adopt the interrupt type and, according to this, perform specific operations. This generates a jitter, due to the time needed to determine the type of interrupt and the routine that treats it.

Currently, for the development of the embedded systems, general purpose processors are used [1]. Nevertheless, these systems can have a non-deterministic behavior, and are not effective in developing real-time systems. Because of these problems, and in order to easily ensure the deadline in the WCET (Worst-Case Execution Time), the developers of embedded systems can oversize the computing needs and use processors with a computing power higher than necessary. The embedded systems can also have a very high consumption in relation to the performance offered by these processors. On the other hand, the increased progress of FPGA devices [1], [2] has enabled the development of specialized controllers which can guarantee meeting the deadlines at low energy rates [3], [4]. Furthermore, it allows the development of SoC (System on Chip) which has the primitives (inter task communication and synchronization) of a real-time operating system implemented in hardware [1] - [6].

The nMPRA architecture is presented in [5]. This architecture tries to solve the aforementioned issues at hardware level. In [7] and [8], the authors defined the first version of the nMPRA, called Multi Pipeline Register Architecture (MPRA). This architecture is based on the MIPS architecture which was changed in order to implement in software the primitives of a RTOS. The initial architecture has a PC register and a set of pipeline registers for each task. These resources are used to save the state of the task when a task context switch operation is performed.

In [5], the MPRA was developed for n tasks and, therefore, the name was changed to nMPRA. This new version is provided with hardware support for static and dynamic scheduling, for unitary handling of events and interrupts and for the RTOS primitives implemented in software (mutexes, semaphores, messages, interrupt handling). Furthermore, this architecture allows interrupts to be attached to the task, and the task can wait for several

This work was supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI-UEFISCDI, project number PN-III-P2-2.1-PED-2016-1473, within PNCDI III. The infrastructure used for this work was partially supported by 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, Sectorial Operational Program for Increase of the Economic Competitiveness co-funded from the European Regional Development Fund.

Digital Object Identifier 10.4316/AECE.2017.03010

interrupts at the same time. It also improves the response time and the predictability of a real-time operating system in the context of time-critical application.

The original contribution of this paper relates to present a solution to improve the response time to interrupts in real time embedded microcontrollers based on the nMPRA. The logic of the nMPRA concerning interrupts consists in allowing their distribution to any of the possible n tasks, inheriting the priority of the attached task ("interrupts as threads"). If multiple interrupts are attached to the same task, the question which arises is what interrupt should be treated first and, if the interrupt is not the first one in the established priority chain (hardware or software), how fast is the control transferred to this interrupt?

Furthermore, this article is structured as follows: Section II presents some comparisons with the RTOS hardware presented in the specialized literature. The nMPRA architecture and the nHSE (Hardware Scheduler Engine) are presented in Section III and Section IV. Section V presents the interrupt behavior in the nMPRA while the proposed enhancements of the interrupt system are presented in Section VI. The conclusions are drawn in Section VII.

II. Related Work

Because nMPRA is a new architecture, the literature of the field has not addressed this issue so far. As a general solution, in [5] a comparison with well-known architectures is made. Thus, we have chosen, as comparison criteria, the following questions (Table 1): if the interrupt controller is specialized or distributed (1); if the interrupts are treated as execution threads (2); if the interrupts can be attached to any task (3); if the interrupt inherits the task priority (4); if the interrupt affects the pipeline (5); if the interrupt requires context software saving (6). There are four main architectures presented in Table 1, namely ARPA-MT [9], PRET [6], [10], [11], Kuacharoen [12], and hthreads [12].

Table I. Comparison Criteria of nMPRA with Four Main Architectures

Features	**nMPRA**	**hthreads [12]**	**ARPA-MT [9]**	**Kuacharoen [12]**	**PRET [6][10] [11]**
(1)	Distributed	Bypass Interrupt Scheduler (CBIS) - specialized	Co-processor (Cop0-MEC)	Specialized	No
(2)	Yes	Yes	-	-	No
(3)	Yes	Yes, but a task can have attached a single interrupt	-	-	No
(4)	Yes	Yes	-	-	No
(5)	No	Yes	-	-	No
(6)	No	Yes	-	-	Yes

Not all these systems have a mechanism for improving the response time to interrupts; they are designed to implement in hardware the directives of a RTOS and to improve the switching time between tasks.

The solution for prioritization and treatment of interrupts presented in this article improved significantly the response time to events.

III. The nMPRA Architecture

Fig. 1 presents the nMPRA architecture. Within this architecture, an instance of the CPU (Central Processing Unit) is named semi CPU for the task *i* (sCPUi). This hardware instance includes its own resources (such as PC registry, general register, pipeline registers and control logic in the Hardware Scheduler Engine (nHSE)), which share resources with other entities (combinational logic allows execution of instructions placed between pipeline registers and the joint nHSE) while task *i* runs task *i* instructions (i = 0,..., n-1).

The sCPU0 is different from others sCPUi because it is the sCPUi unit active after reset and that can activate the others sCPUi units. Furthermore, this unit has the highest priority in the system and has access to the configuration and the monitoring register associated to each sCPUi or to nHSE.

The nMPRA has two schedulers, one for scheduling tasks with static priorities and one for scheduling tasks with dynamic priorities, both being preemptive. The sCPU0 is the priority task in the system and its priority cannot be changed even by the dynamic scheduler. It also performs the selection of the active scheduler when the application is starting execution. The schedulers can achieve fast switching between tasks (1-3 processor cycles) at the occurrence of external or internal events. This architecture can allow a task to wait for several types of events using a single instruction. In the case of the dynamic scheduler, it is allowed to change task priority using a single register attached to each task (except for the sCPU0 task). This enables the implementation of various scheduling policies, depending on application requirements.

The nMPRA architecture does not have a specialized hardware interrupt controller; instead it allows the attachment of hardware interrupts and of events to the tasks in the system (an interrupt or event can be attached to a single task). Tasks must execute a single instruction in order to get in line to wait for the attached events and interrupts. Furthermore, the interrupts may be attached to another task during application execution.

The noticeable fact about the nMPRA architecture, shown in Fig. 1, is that each sCPUi has its own set of pipeline registers (ID / EX, MEM / WB, EX / MEM, IF / ID), its own set of general registers, a Program Counter (PC), and a set of special registers used by the nHSE for planning and treating interrupts and events. Because of these characteristics, task switching can be performed in less than 1.5 processor cycles or 3 processor cycles, if the currently running task executes a memory work instruction. The other architecture resources are the ones shared by the sCPUi.

IV. The nHSE Architecture

The nHSE architecture is shown in the Fig. 2. It is actually a hardware block within a microprocessor that implements the primitives of a real-time operating system. As input, it has all the events (message, deadline, interrupts, timer, etc.) which can determine the change of the running task and, as output, it generates a signal that can validate a single sCPUi to enter or continue execution.

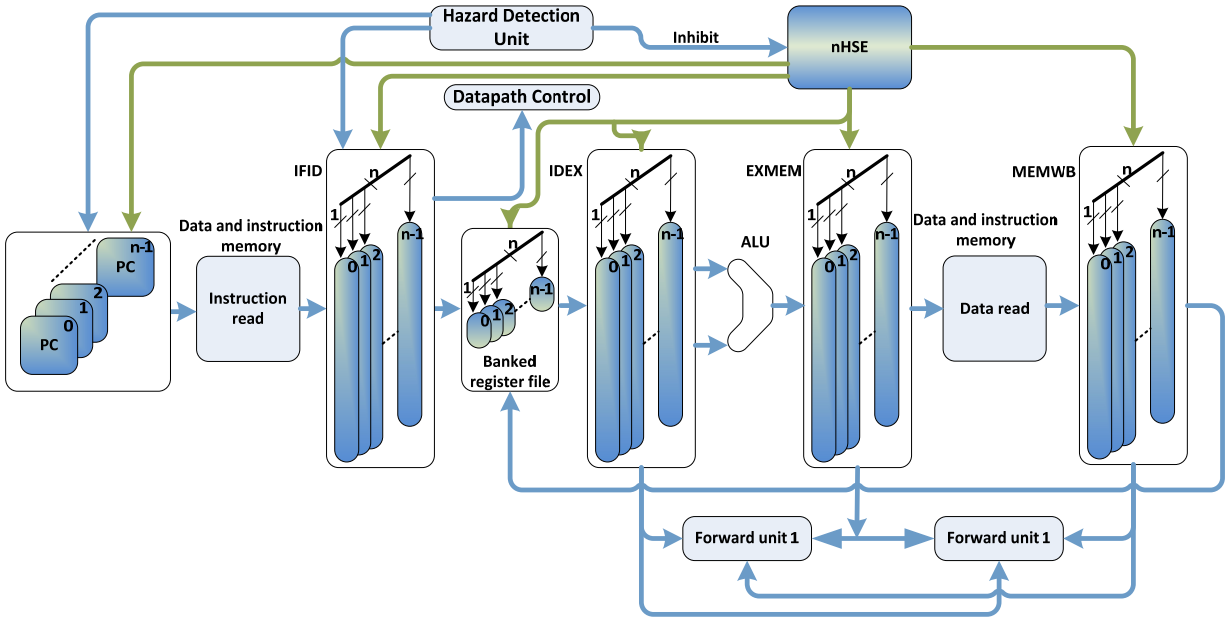

Figure 1. The nMPRA architecture: PC – program counter, IFID – Instruction Fetch Instruction Decode stage, ID/EX – Instruction decode–execute stage, EX/MEM – execute-memory stage, MEM/WB – memory-write back stage [5]

The identifier from the ID register selects the active sCPUi if there is an active event. Otherwise the system is idle and will come out of this state when an event occurs. If the sCPUi associated task, which experienced the event, clears it, the task stops itself. If its execution is to be continued, the task must enable the self-sustaining event.

The solution has some drawbacks such as the limited number of possible levels of nesting, which is limited to the number of sCPUi. Another disadvantage is the lack of interrupt handler vectorization. If more than one interrupt is attached to a sCPUi, their order of treatment is determined by software, which may introduce additional delays. The logic of events at each sCPUi, and the level of each sCPUi are shown in Fig. 3.

The scheduler is constantly monitoring the events addressed to the sCPUi. The possible sCPUi events are: timer interrupts (TEvi), watchdog timer (WDEvi), two interrupts used for preventive signaling of the deadline (D1Evi and D2Evi), attached interrupts (IntEvi), mutexes used for handling shared resources (MutexEvi), synchronization and inter-task communication events between sCPUi (SynEvi), self-sustaining execution information for the current sCPUi (lr_run_sCPUi). The events can be validated with lr_enTi, lr_enWDi, lr_enD1i, lr_enD2i, lr_enInti, lr_enMutexi and lr_enSyni signals (see Fig. 3 and Fig.4a). There is one exception, namely lr_run_sCPUi.

Each sCPUi has a register called Event Register (EVi) which allows reading active events as shown in Fig.4c without sCPUi blocking. The instructions proposed for the TRi and EVi registers are: *movcr TRi, Rj; movcr Rj, TRi movcr EVi, Rj; movcr Rj, EVi*.

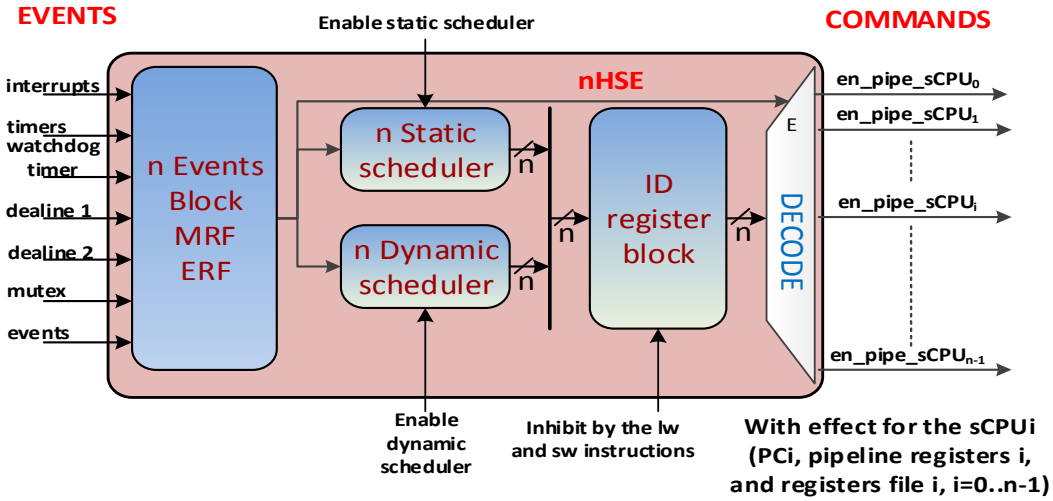

Figure 2. The nHSE architecture [5]

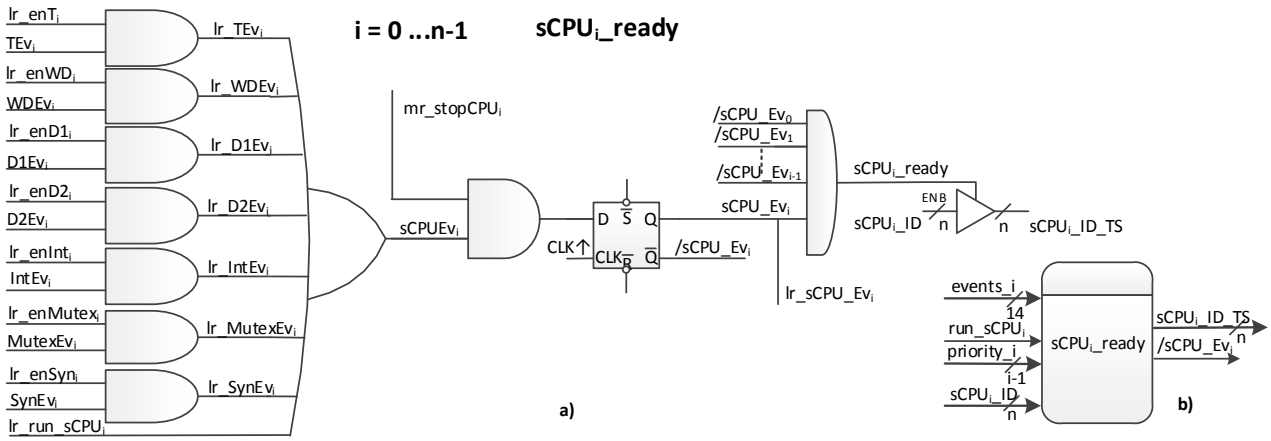

Figure 3. The sCPUi level hardware scheduler (block of nHSE) – (a) digital logic for ready state, (b) block diagram [5]

31..8	7	6	5	4	3	2	1	0
0	lr_run_sCPUi	lr_enSyni	lr_enMutexi	lr_enInti	lr_enD2i	lr_enD1i	lr_enWDi	lr_enTi

a)

31..8	7	6	5	4	3	2	1	0
0	lr_run_sCPUi	lr_SynEvi	lr_MutexEvi	lr_IntEvi	lr_D2Evi	lr_D1Evi	lr_WDEvi	lr_TEvi

b)

31..8	7	6	5	4	3	2	1	0
0	lr_run_sCPUi	SynEvi	MutexEvi	IntEvi	D2Evi	D1Evi	WDEvi	TEvi

c)

Figure 4. The structure of the: a) TRi register, b) Rj register at the return from the wait instruction, c) EVi register

V. INTERRUPTS IN nMPRA

The model suggested in [5] is similar to the "interrupts as threads" approach. The model is illustrated in Fig. 5. The interrupts in nMPRA are treated as events attached to the real-time executive or to tasks thus borrowing the priority of the tasks they are attached to. Assuming that there are p interrupts in the system for each interrupt, a global register is provided (accessible to all sCPUi), called *INT_IDi_register* with n1 useful bits; this register allows the storage of the tasks IDs which are attached to the interrupt.

The enabling of the *INTi* interrupt (Fig. 5) validates the DEMUX demultiplexer which will activate one of the *INT_i0, ..., INT_in-1* signals. The OR gate (see Fig. 5) can collect all interrupts in the system. They can be attached to sCPUi, if all the *INT_IDi_register* (i = 0 ... p-1) global registers are written with *i* value. The D flip-flop is designed to synchronize the random occurrence of the *INTi* event producing the *IntEvi* signal (Fig. 3).

Fig. 6 shows how the proposed and implemented solution on Virtex7 works. In this case, 4 sCPUs are presented with *TRi* registers and 4 external interrupts. The values in Fig. 6 are expressed as hexadecimal numbers. The 4 interrupts are attached as follows: 0 to task 0, 1 to task 1, 2 to task 2, and 3 to task 3. At moment T1 interrupt 1 appears attached to task 1. Assuming that task 0 also expects an interrupt, task 1 is released in execution and the value of *crEV1* register is 10H (the value of bit 4 from Fig.4c, named *IntEvi*, is 1). At moment T2 the interrupt attached to task 3 appears/occurs; if tasks 0 and 1 are suspended, task 3 is being launched in execution and the value of *crEV3* register is 10H (the value of bit 4 from Fig.4c, named *IntEvi*, is 1). At moment T3 time, the interrupt attached to task 0 occurs, and no matter what task is running, task 0 is launched in execution and the value of *crEV0* register is 10H (the value of bit 4 from Fig.4c, *IntEvi*, is 1).

The nMPRA architecture has some disadvantages, such as the fact that the nested level of the interrupts is limited to the number of tasks and that there is no interrupt handler vector. If more interrupts are attached to the same sCPUi, and if they occur simultaneously, then the software establishes the order in which the interrupts will be handled; this can lead to additional delays.

Instruction *wait* validates the expected events and its format is instruction *wait*. The instruction format of *wait* is *wait Rj*. It blocks the pipeline until the event selected by setting *Rj* register occurs. The *Rj* register is automatically transferred to the task register (TR). Each sCPUi has a *TRi* register with a structure shown in Fig.4a. A more efficient method involves the use of mnemonics which imply an immediate value (expected events) in the instruction body, in the following form: *wait Rj, events*. The events expected by the *wait* instruction are loaded in the *TRi* register and, during return, the expected and occurred events are loaded into the *Rj* register as shown in Fig.4b.

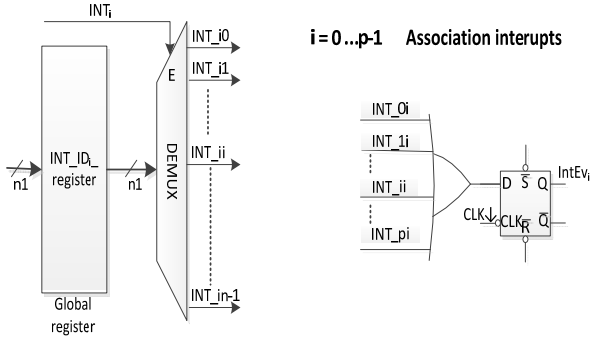

Figure 5. Association of the interrupts with the sCPUi (task i) [5]

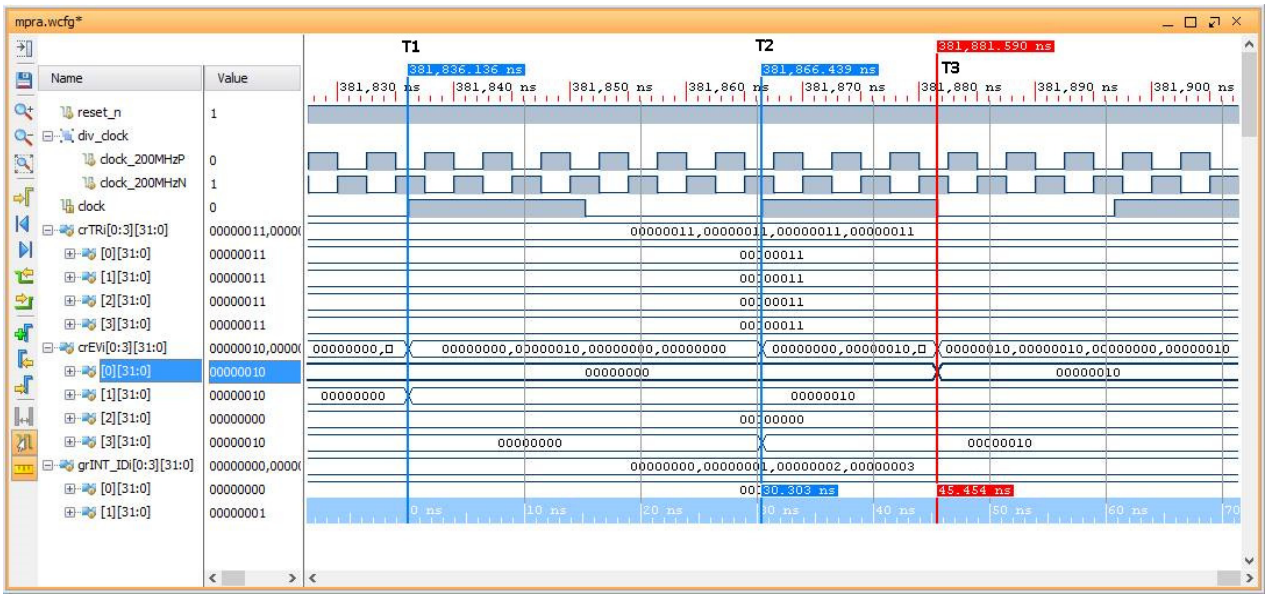

Figure 6. The implementation of the proposed solution on Virtex 7

As specified in [5],[7],[8], and in [14] - [24], this architecture has several interesting features, such as: it does not use a specialized interrupt controller, the interrupts inherit the priority of the tasks (sCPUi), a task can attach none, one, several or even all the p interrupts in the system, the priority of the interrupts that are attached to the same task is established by the programmer, an interrupt that is attached to a task can interrupt a lower priority task but cannot interrupt the execution of the task to which it is attached, or a higher priority task, one interrupt can be attached only to a single task, the interrupt can be a task, all the interrupts can be attached to a single task.

VI. Enhanced Interrupt Response Time in the nMPRA

A first improvement concerning the response time of interrupts, without taking into account other types of events, is proposed in [14]. This article examines the response time of interrupts when other events, such as mutexes, semaphores, messages, interrupt handling, are taken in consideration. Furthermore, the original contribution and scientific merit of this paper by comparing to other scientific papers [5],[7]-[8],[14]-[24], is a solution, obtained from the synthesis with ChipScope analyzer, for improving the handling time of interrupts when the other types of events are active and validated.

An extreme scenario is the one in which a sCPUi waits for all 7 types of events. The question is: which event is handled first and what is the order of event handling? An initial answer can be provided by the application programmer, depending on the objectives the task has to accomplish. A simple solution would be to prioritize the events through the software. An extremely possible scenario is shown in Program1. This extreme scenario shows that even though the occurrence of an event is associated with the highest priority task in the system, it switches the task to the next machine cycle (unless memory access instructions are run). The event handling can be delayed, especially in the unfortunate situation when the expected events occur simultaneously and when the task expects all of them.

The Program 1 presented above is a software solution for the prioritization of the events and the interrupts associated with a task. The proposed solution is versatile because it requires no additional hardware support and allows the easy modification of the program priorities. The major disadvantage, however, is the delay introduced for low priority events. Even if it is the only event which occurs, the low-priority event has to wait for all tests meant to identify the higher priority events to be completed; this may cause a delay that cancels the hardware speed of task switching. Assigning events (and therefore interrupts) to task *i* in a system with *i* tasks must be done carefully in order to preserve system performances.

A first improvement in the response time to the occurrence of interrupts and their prioritization [14] regards the use of a priority encoder at each sCPUi level, as shown in Fig. 7. Depending on the total number of interrupts, each sCPUi is has a register with $[\log_2 pi]$ bits. This register provides the number of the highest priority interrupt, in case more than 2 simultaneous interrupts occur, or when more interrupts are attached to the same sCPUi. The truth table for the priority encoder is shown in Table 2. In such cases, the *IntEvi* procedure becomes like in Program 2. This solution requires additional hardware support for each sCPUi. The advantages of this solution include:

- The start time of the interrupt handler is the same for all interrupts, if at one point a single interrupt is active. In the case of the software solution, this time depends on the position of the interrupt in the decision chain (further the test done by the *IntEvi* software is from the start of the procedure, the greater delay).
- The procedure for solving the cause for the interrupt is faster when dealing with a small number of interrupts which occur simultaneously.

TABLE II. The Priority Encoder

Most	**Interrupts for sCPUi**				**Least**	**IntRegi**			
INT_0i	INT_1i	…	INTii	…	INTpi	$[\log_2 pi]$	…	b1	b0
0	0	0	0	0	0	0	0	0	0
1	x	x	x	x	x	0	0	0	1
0	1	x	x	x	x	0	0	1	0
…	…	…	…	…	…	…	…	…	…
0	0	0	0	0	1	1	1	1	1

Program 1 – pseudo-code for the software prioritization of the events and the interrupts associated with a task

```
sCPUi_taski:
        Initializing the task i

main_sCPUi_loop:
        Load Rj register with all events and interrupts
        Waits all sCPUi possible events and interrupts
       Take the events and interrupts occurred
        Select Inti interrupts
        Test if at least one interrupt occurred (if it is active)
        Jump if no interrupt occurred (jump test_D2Evi )
        Call interrupt handler procedure (Call IntEvi)

test_D2Evi:
        Take the events
        Select D2Evi event
        Test if the event it is active
        Jump if the D2Evi event it is not active (jump test_D1Evi)
        Call event handler procedure (call D2Evi)

test_D1Evi:
        ……..    ……..

test_TEvi:
        ……..    ……

test_WDEvi:
        ……..    ……

test_MutexEvi:
        ……..    ……

test_SynEvi:
        // Main body of the task
        …….     ……..
        Jump    main_sCPUi_loop

IntEvi:
        // The procedure for interrupt
        Save the interrupts
        Take base address of the peripheral
        Take peripheral status for the higher priority interrupt
        Take the interrupt i1 status
        Test if interrupt i1 it is active
        Jump if interrupt i1 is not active  (jump test_inti2)
        Call interrupt handler procedure for the most priority interrupt i1 (call int_handler_i1)

test_inti2:
        ……..    ……

test_intipi1:
        Test the least important interrupt ip1
        ……..    ……
        Return from procedure

D2Evi:
        // The procedure for the D2Evi event
……      ……

SynEvi:
        // The last and the least priority event procedure for SynEvi
……      ……

int_handler_i1:
        // Interrupt handler for i1 the most priority interrupt
……      ……..

Int_handler_ip1:
        // Interrupt handler for ip1 the least priority interrupt
……      ……
```

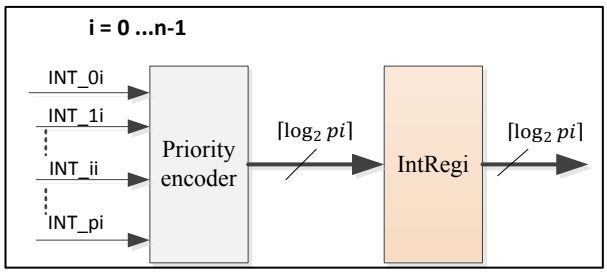

Figure 7. The priority encoder

Program 2 - pseudo-code for the prioritization of the events and the interrupts associated with a task using a hardware priority encoder

```
jIntTable:
        IntH0, IntH1, …, IntHp1i
        // Interrupts table with p1 =< pi
IntEvi:
        // Starting the procedure for interrupt
        Save the events
loop_IntEvi:
        Select Inti interrupts
        Test if at least one interrupts occurred
        Jump if no interrupt occurred (jump
              exit_IntEvi)
        Take the most priority interrupt that it is active
        Call interrupt handle (call IntH0)
        Take actual interrupts
        Loop to test if exist more interrupts (jump
             loop_IntEvi)
exit_IntEvi:
        Return from procedure
……..
IntH0:
        // First interrupt most priority one
        Jump to interrupt 0 handler procedure
IntH1:
        // The second interrupt the next most priority
        Jump to interrupt 1 handler procedure
……..
IntHp1:
        // The last interrupt least priority one
        Jump to interrupt p1 handler procedure
……..
interrupt_handler0:
        TODO
……..
        Return from interrupt handler
interrupt_handlerp1i:
        TODO
……
        Return from interrupt handler
```

The disadvantages of the solution (presented in Program 2) are that requires additional hardware for each sCPUi and the priorities are fixed at sCPUi level (*int_0i* has the highest interrupt priority and *int_pi* has the lowest interrupt priority). The priority is transferred to the sCPUi, according to the interrupts attached. A second improvement, which requires additional hardware, promotes the idea that once the interrupt-type events gain the highest priority, the transfer of the control to the interrupt handler would be made directly by loading the PC with the start address of the interrupt handler. In this case, the return instruction from the interrupts will either return to the interrupt program, if there are no more active interrupts, or it will return to the starting address of the interrupt handler associated to the next priority interrupt attached to a sCPUi. The block diagram is illustrated in Fig. 8. The main problem of this solution is the generation of the *Dec_En signal*. The pseudo code for generating this signal is presented in Program 3. This hardware solution is designed to eliminate the delays introduced by the loop in Program 2 (about 10 instructions). As one of the advantages, this solution improves the response time to interrupts, from the moment when control is transferred to the interrupt handler. Returning from an interrupt, the handler switches quickly, through the modification of *jr $ ra*'s behavior instruction, to the next interrupt handler if there are active interrupts for sCPUi. The disadvantage of this solution is that it requires additional hardware resources (address decoder, registers with trap cells and the generating logic for the *Dec_En signal*).

Program 3 - pseudo code for Dec_En signal

It is called the IntEvi procedure
IntEvi:
Repeat while they are active interrupts
Enable the scheme for automatic jump to an interrupt handler (Dec_En = 1)
The content of the registers that contain the cell trap for the interrupt handler corresponding to the higher priority interrupt is taken and transferred to the PC.
Disable the Dec_En = 0, and transfers control to the interrupt handler
Execute the interrupt handler routine
If the *jr $ra* instruction is executed and there are no longer active interrupts
(IntEvi = 1)
then Exit
Exit (execute normal *jr $ra* with return from *IntEvi* routine)

VII. Conclusion

The nMPRA has a very good switching time between tasks, when an event associated with a higher priority task occurs. Because the nMPRA allows simultaneous synchronization for up to seven events, a major issue is the time in which events can be treated if they occur all at the same time. The nMPRA architecture may become less efficient if there are no low cost solutions found for this problem. This article attempted to perform an analysis of this issue in the case of interrupts and only the interrupt routine has been taken into account. The last solution proposed is a very high speed one (low response time) but which implies a significant consumption of hardware resources. The proposed solution interrupts the sCPUi if the *IntEvi* procedure is active. From the author's point of view, this is an improvement of the sCPUi behavior to interrupts.

Acknowledgment

This work was supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI-UEFISCDI, project number PN-III-P2-2.1-PED-2016-1473, within PNCDI III. The infrastructure used for this work was partially supported by 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, Sectorial Operational Program for Increase of the Economic Competitiveness co-funded from the European Regional Development Fund.

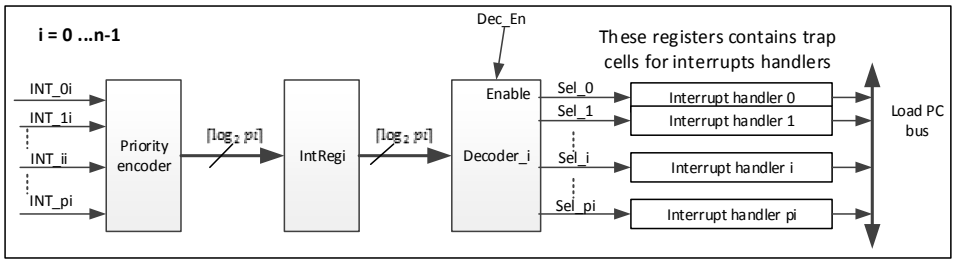

Figure 8. The priority encoder

REFERENCES

[1] J. Shawash, D. R. Selviah, "Real-Time Nonlinear Parameter Estimation Using the Levenberg–Marquardt Algorithm on Field Programmable Gate Arrays," IEEE Trans. Ind. Electron., vol. 60, no. 1, pp. 170–176, Jan. 2013. doi:10.1109/TIE.2012.2183833.

[2] M. Shahbazi, P. Poure, S. Saadate, M. R. Zolghadri, "FPGA-Based Reconfigurable Control for Fault-Tolerant Back-to-Back Converter Without Redundancy," IEEE Trans. Ind. Electron., vol. 60, no. 8, pp. 3360–3371, Aug. 2013. doi:10.1109/TIE.2012.2200214.

[3] M. Shahbazi, P. Poure, S. Saadate, M. R. Zolghadri, "Fault-Tolerant Five-Leg Converter Topology With FPGA-Based Reconfigurable Control," IEEE Trans. Ind. Electron., vol. 60, no. 6, pp. 2284–2294, Jun. 2013. doi:10.1109/TIE.2012.2191754.

[4] T. T. Phuong, K. Ohishi, Y. Yokokura, C. Mitsantisuk, "FPGA-Based High-Performance Force Control System With Friction-Free and Noise-Free Force Observation," IEEE Trans. Ind. Electron., vol. 61, no. 2, pp. 994–1008, Feb. 2014. doi:10.1109/TIE.2013.2266081.

[5] V. G. Gaitan, N. C. Gaitan, 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.

[6] M. Zimmer, D. Broman, C. Shaver, E. A. Lee, "FlexPRET: A processor platform for mixed-criticality systems," in 20th IEEE Real-Time and Embedded Technology and Applications Symposium - RTAS, pp. 101–110, Apr. 2014. doi:10.1109/RTAS.2014.6925994.

[7] E. Dodiu, V. G. Gaitan, A. Graur, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – architecture description", in IEEE 35'th Jubilee International Convention on Information and Communication Technology, Electronics and Microelectronics, Croatia, pp. 859-864, 24 May 2012. INSPEC Accession Number: 12865464.

[8] E. Dodiu, V. G. Gaitan, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – concept and theory of operation," in IEEE EIT International Conference on Electro-Information Technology, Indianapolis, USA, pp. 1–5, May 2012. doi:10.1109/EIT.2012.6220705.

[9] A. S. R. Oliveira, L. Almeida, A. B. Ferrari, "The arpa-mt embedded smt processor and its rtos hardware accelerator", IEEE Trans. Industrial Electronics, vol. 59, no. 3, pp. 890-904, August 2009. doi:10.1109/TIE.2009.2028359.

[10] I. Liu, J. Reineke, D. Broman, M. Zimmer, E.A. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," Computer Design (ICCD), 2012 IEEE 30th International Conference on, vol., no., pp.87-93, Sept. 30 2012-Oct. 3 2012, doi:10.1109/ICCD.2012.6378622.

[11] S.A. Edwards, E.A., Lee, "The Case for the Precision Timed (PRET) Machine," Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE, vol., no., pp.264-265, 4-8 June 2007.

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

[13] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, R. Sass, "hthreads: a hardware/software co-designed multithreaded RTOS kernel", Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on, vol.2, pp.338, 19-22 Sept. 2005. doi:10.1109/ETFA.2005.1612697.

[14] N. C. Gaitan, V. G. Gaitan, E.-E. (Ciobanu) Moisiuc: "Improving Interrupt Handling in the nMPRA", In Development and Application Systems (DAS), 2014 International Conference on. IEEE, pp. 11-15, 15–17 May, 2014. doi:10.1109/DAAS.2014.6842419.

[15] C. Kyrkou, T. Theocharides, "A Parallel Hardware Architecture for Real-Time Object Detection with Support Vector Machines," in IEEE Transactions on Computers, vol. 61, no. 6, pp. 831-842, June 2012. doi:10.1109/TC.2011.113.

[16] N. C. Gaitan, I. Zagan, V. G. Gaitan, "Predictable CPU Architecture Designed for Small Real-Time Application - Concept and Theory of Operation," International Journal of Advanced Computer Science and Applications – IJACSA, vol. 6, no. 4, 2015. doi:10.14569/IJACSA.2015.060406.

[17] L. Andries, G. Gaitan, "Dual priority scheduling algorithm used in the nMPRA microcontrollers: Subtitle as needed (paper subtitle)," 2014 18th International Conference on System Theory, Control and Computing (ICSTCC), Sinaia, 2014, pp. 43-47. doi:10.1109/ICSTCC.2014.6982388.

[18] E. E. C. Moisuc, A. B. Larionescu, I. Ungurean, "Hardware event handling in the hardware real-time operating systems," 2014 18th International Conference on System Theory, Control and Computing (ICSTCC), Sinaia, 2014, pp. 54-58. doi:10.1109/ICSTCC.2014.6982390

[19] I. Zagan, V. G. Gaitan, "Improving the Performances of the nMPRA Processor using a Custom Interrupt Management Scheduling Policy," Advances in Electrical and Computer Engineering, vol.16, no.4, pp.45-50, 2016, doi:10.4316/AECE.2016.04007.

[20] E. E. Moisuc, A. B. Larionescu, V. G. Gaitan, "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] A. Kalyansundar, R. Chattopadhyay, "A Novel Approach to Hardware Architecture Design and Advanced Optimization Techniques for Time Critical Applications," 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, Shanghai, 2008, pp. 9-15. doi:10.1109/EUC.2008.113.

[22] I. Zagan, V. G. Gaitan, "Schedulability analysis of nMPRA processor based on multithreaded execution," 2016 International Conference on Development and Application Systems (DAS), Suceava, 2016, pp. 130-134. doi:10.1109/DAAS.2016.749256.

[23] L. Andries, V. G. Gaitan, E. E. Moisuc, "Programming paradigm of a microcontroller with hardware scheduler engine and independent pipeline registers - a software approach," 2015 19th International Conference on System Theory, Control and Computing (ICSTCC), Cheile Gradistei, 2015, pp. 705-710. doi:10.1109/ICSTCC.2015.7321376.

[24] I. Zagan, V. G. Gaitan, "Improving the Performances of the nMPRA Processor using a Custom Interrupt Management Scheduling Policy," Advances in Electrical and Computer Engineering, vol.16, no.4, pp.45-50, 2016, doi:10.4316/AECE.2016.04007.