Improving Interrupt Handling in the nMPRA

Nicoleta Cristina GAITAN, Vasile Gheorghita GAITAN, Elena-Eugenia (CIOBANU) MOISUC
Faculty of Electrical Engineering and Computer Science
Stefan cel Mare University of Suceava
Suceava, Romania
cristinag@eed.usv.ro, gaitan@eed.usv.ro, neli.ciobanu@eed.usv.ro

***Abstract*—In the most Real Time Operating Systems (RTOS), the interrupt handlers are implemented in software and they can increase the response time to external events and the overload of the CPU. Therefore, the newest RTOSs implement in hardware the interrupt handlers in order to eliminate these two problems. By analyzing traditional models for the management of interrupts, we can emphasize their inability to provide the temporal determinism required in real-time systems. This paper presents an interrupt handler implemented in hardware based on a method that uses a unified space of priorities for the tasks and interrupts, so there is not a specialized interrupt controller. This solution is integrated in the MPRA (Multi Pipeline Register Architecture) processor that contains a hardware RTOS. The major difference compared to other architectures with hardware scheduler is that the MPRA is a multi-pipeline architecture, which means that each task has its own set of pipeline registers.**

Keywords— hardware scheduler; interrupt; microprocessor; pipeline register; real time system

I. INTRODUCTION

One of the fundamental requirements of RTS (*Real-Time Systems*) is the determinism of the critical real-time tasks. The execution rate of the tasks, the overload generated by the operating system, and time spend for the switching operations of the tasks' context are just a few parameters that can generate jitters and deadline misses in RTS based on software schedulers.

In this paper, we propose an interrupt system for a real-time scheduler implemented in hardware, which eliminates the overload generated by the operating system and context switching operations. The novelty is the method used for the selection of the tasks and the management of the interrupt system that does not require a dedicated interrupt controller.

The research presented in this paper is based on the functional Multi Pipeline Register Architecture (MPRA) processor presented in [1] and [2], which provides a very low time for the context switching operations as a consequence of the architecture concepts. This processor is capable to perform automatic context switching and to start the new task in a range of 1 to 3 clock cycles. The processor implements a structure with multiplexed resources for pipeline registers and registers file and relies on an integrated hardware scheduler, which can run its own scheduling algorithms, such as EDF (*Earliest Deadline First*) or RMA (*Rate Monotonic Algorithm*).

The hardware scheduler is integrated into the processor, and the context switching operations requires the remapping of the registers file and of the pipeline registers file, both actions being driven by the HSE unit (*Hardware Scheduler Engine*) [3].

The MPRA provides a dynamic mechanism for the management of the interrupts. In the MPRA, the interrupts are treated as tasks, and in order to guarantee a robust scheduling scheme and to eliminate the priority inversions [4], a proper assignment of priorities and an accurate assessment of the problem are needed. In the most applications based on microcontroller, a priority rule allows that all events to be captured and treated. Because of this fixed rule, which does not change during the execution of the application, there may be some low-priority interrupts attached to the low-priority tasks that interrupt the higher-priority tasks, causing the increase of their jitters. In order to eliminate this issue, the MPRA assigns priorities to interrupts from a common group of tasks.

FPGA (*Field Programmable Gate Array)* devices offer integrated elements with complexities of the application oriented integrated circuits (ASIC - *Application Specific Integrated Circuit*), with the advantage of programmability or better said, of configurability [5]. The FPGA devices allow the design of specialized hardware architectures with the advantage of the flexibility programmable environment that can be implemented [6]. Based on these devices, hardware supports for RTOS primitives that can be easily implemented [7] [8]. In this paper, we propose a hardware support for predictable handling of the interrupts.

This article is structured as follows: in section II is presented the *nMPRA* (multiplied by *n* times), in section III is presented the *nHSE* architecture including an interrupt management (subject of this article and the novelty for the proposed architecture), and in section IV conclusions are presented.

II. THE NMPRA

The *nMPRA* architecture is presented in Fig. 1. The context switching operations can be achieved in one processor cycle, and the response to an external event is delayed up to 1.5 processor cycles because each task has a set of pipeline

978-1-4799-5094-2/14/$31.00 ©2014 IEEE

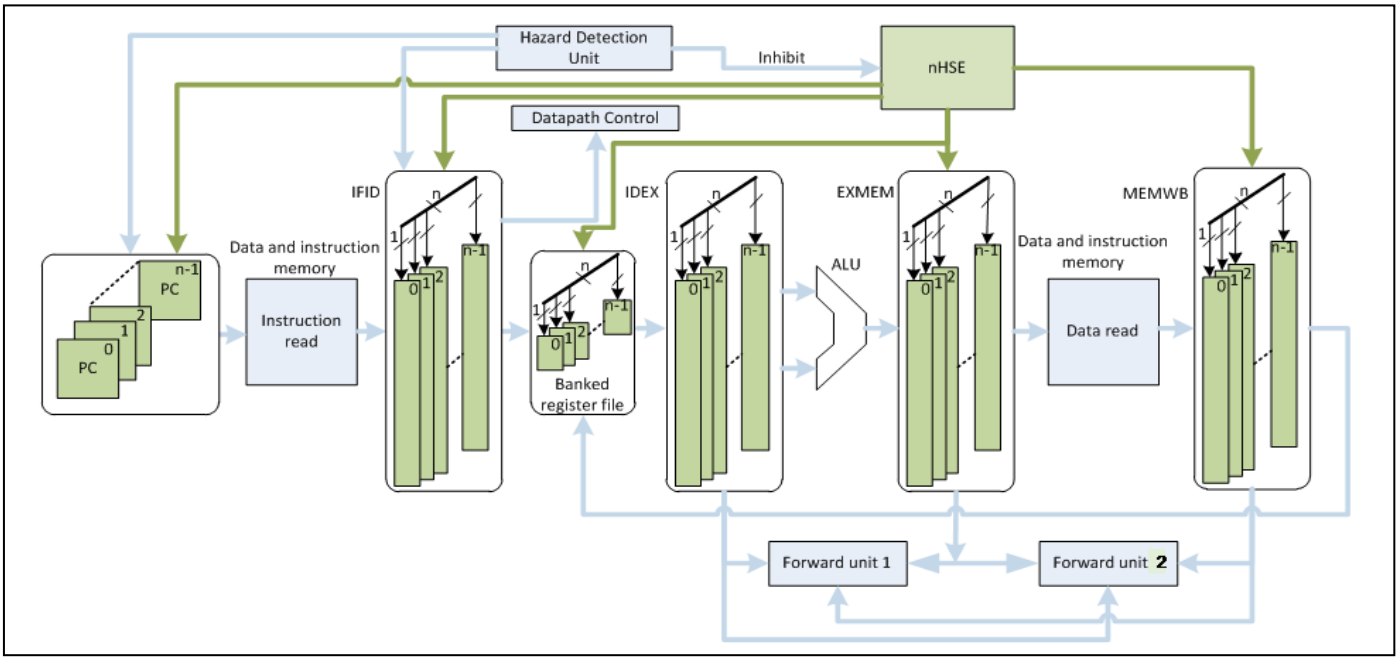

Fig. 1. The nMPRA. PC-program counter, IFID-Instruction Fetch Instruction Decode stage, IDEX-Instruction decode-execute stage, EXMEM-execute-memory stage, MEMWB-memory-write back stage (source: [3]).

register and a set of general-purpose registers. For these reasons, the architecture is very fast. All tasks share the other resources. An instance of a task is called *semi CPU* for the task *i* ($sCPU_i$) and handles a single task.

There is not a specialized controller for the interrupts, but the nMPRA provides a distributed structure that permits to assign an interrupt to an RTOS task. The programmers can change the priority of the interrupts by assigning them to another task [3].

III. NHSE ARHITECTURE AND INTERRUPTS HANDLING

The *nHSE* (Fig. 2) has input for events (interrupts, deadline, watchdog timers, timers, mutexes, message events, self-support event execution, as well enabling signals of static and dynamic schedulers and inhibiting the execution of load and store instructions) used to generate the $sCPU_i$ activation signals [3].

The nHSE architecture is illustrated in Fig. 3. In our proposed solution, the interrupts are events that can be assigned to the real-time kernel or tasks. The kernel must be a real-time monitor type. Once entered into the monitor, this cannot be interrupted (interrupts are disabled). As a result, the monitor functions must be short, without interactions and idles. The interrupts attached to a task can interrupt only strictly lower priority tasks, which are into the running state.

Suppose that there are *p* interrupts in the system. For each interrupt of the system, there is a global register, named $INT\_ID_i\_register$, with *n* useful bits that store the ID of the task associated to the interrupt. The enabled interrupt INT_i (Fig. 3) validates the demultiplexer DEMUX which, in turn, will activate one of the signals INT_i_0 ... INT_i_{n-1}. *OR* gate (Fig. 3) can collect all interrupts of the system. All interrupts may be attached to $sCPU_i$ if all *p* registers $INT\_ID_i\_register$ (*i = 0, ..., p-1*) are set with the value *i*. Likewise, neither a interrupt can be attached to $sCPU_i$ if none of the *p* registers $INT\_ID_i\_register$ (*i = 0, ..., p-1*) is not written with value *i*.

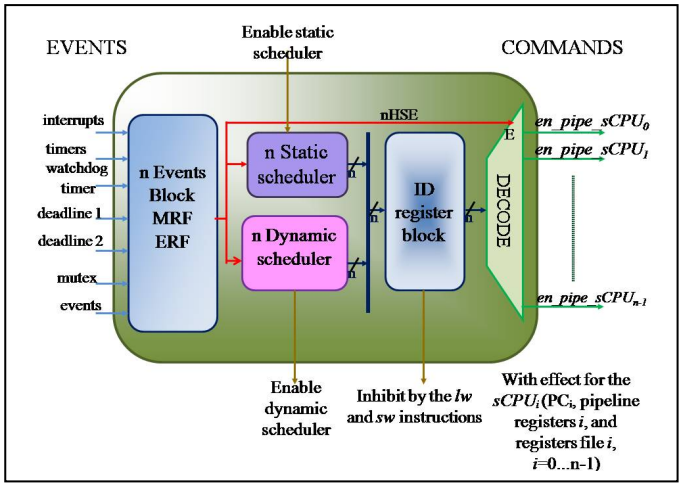

Fig. 2. The nHSE Architecture (source: [3]).

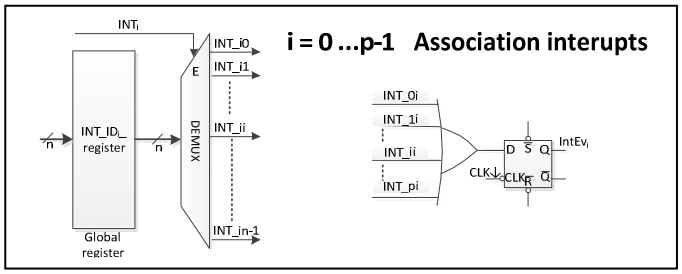

Fig. 3. Association of the interrupts to the $sCPU_i$ (task i) (*source*: [3]).

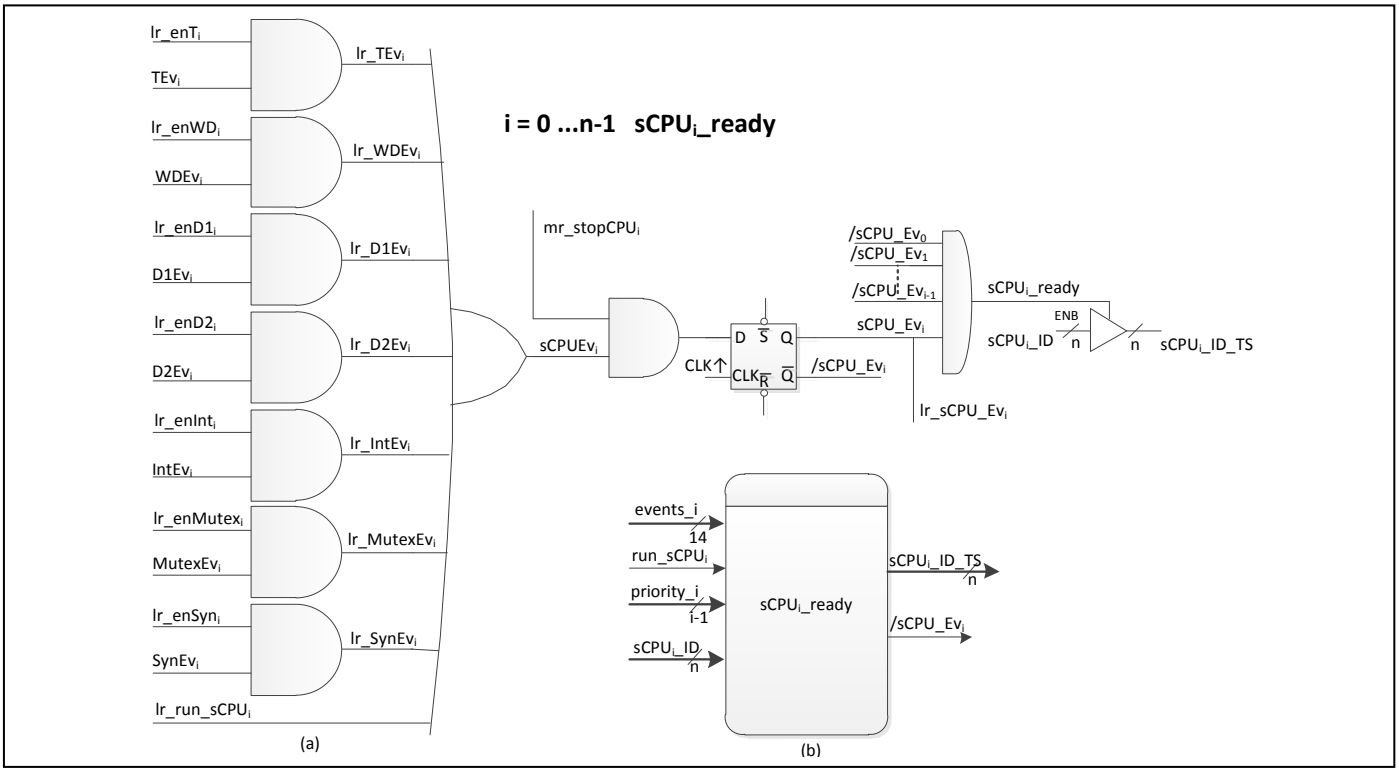

Fig. 4. The $sCPU_i$ level hardware scheduler (block of nHSE) – (a) digital logic for ready state, (b) block diagram (*source*: [3]).

D flip-flop synchronizes the random occurrence of events such as interrupt INT_i producing event $IntEv_i$ (Fig. 4) and it is accounted on the falling edge of the system clock.

This proposed scheme has some strong and interesting characteristics: there are not a specialized interrupt controller; interrupts inherit the priority of the tasks ($sCPU_i$); a task can have attached none, one, several, or all the *p* system interrupts; all interrupts attached to the same task have the same priority; an interrupt attached to a task can interrupt only the lowest priority tasks; an interrupt may be assigned to a single task; an interrupt can be seen as a task; all interrupts can be assigned to a single task; an interrupt does not reset the pipeline of other $sCPU_i$; it does not require the saving and restoring operations of the context; interrupts can be nested.

We assume that the devices that generate the interrupts have a bit to signal the interrupt condition, a bit for the interrupt validation, and a bit for the interrupt clearing. Analyzing the scheme of the interrupts, we noticed the following problem: what happens if all interrupts are attached to the same $sCPU_i$ and occur simultaneously? In this article, we present two solutions to this problem:

1. The software solution is shown in Fig. 5. It is simple (it does not require additional hardware modules) and versatile because the priorities of interrupts can be easily changed. A disadvantage is the delays introduced by the test blocks and interrupt handling routines in the case when more interrupts are attached the same $sCPU_i$ and they occur simultaneously. In addition, the delay generated by the test blocks depends on the moment position of the test blocks.
2. Another solution involves an additional hardware block as shown in Fig. 6.

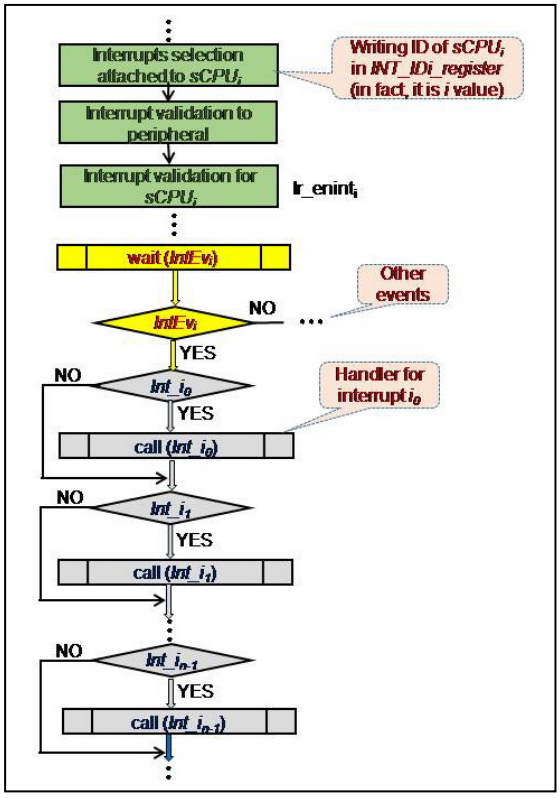

Fig. 5. The software solution.

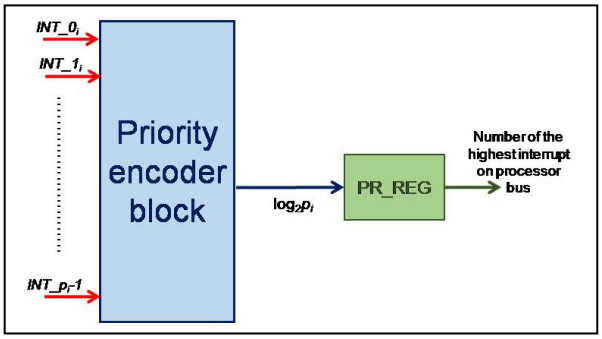

Fig. 6. Additional hardware block

At the occurrence of one or more interrupts, the priority encoder block will generate the appropriate number of the highest priority interrupt. This number is multiplied by 4 to calculate the displacement of a grid cell-trap disruption (Fig. 7). The interrupt handler address is read and the control is transferred to the interrupt handler. As a consequence, for each interrupt, the delay time of decision block will be the same.

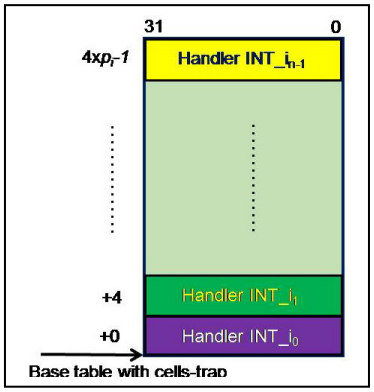

Fig. 7. The priority encoder block (cells-trap)

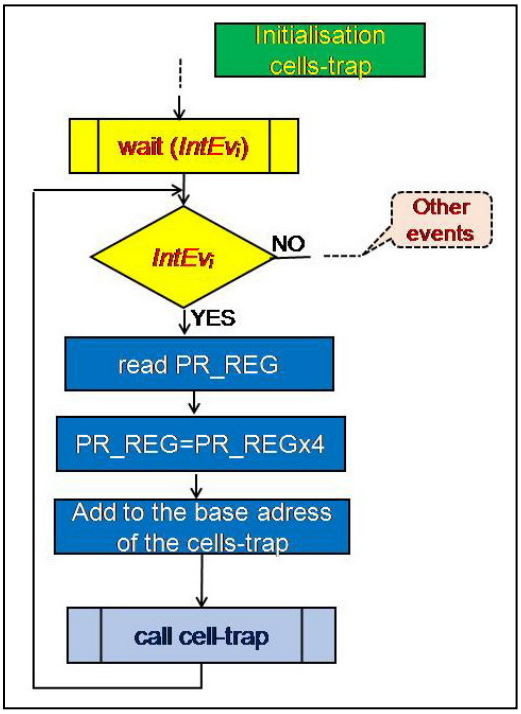

Fig. 8. The hardware solution.

The solution is fast (Fig. 8), but requires an additional hardware block, whose complexity is given by the total interrupts of the processor. An example is shown in Fig. 9 that contains the equations (a), truth tables (b) and logical scheme (c) for a priority encoder with $p_i = 4$ (INT_0_i is the highest interrupt).

$$pr\_1 = \overline{INT\_0_i} \cdot \overline{INT\_1_i}$$
$$pr\_0 = \overline{INT\_0_i} \cdot INT\_1_i + \overline{INT\_0_i} \cdot \overline{INT\_2_i} \qquad \text{(a)}$$

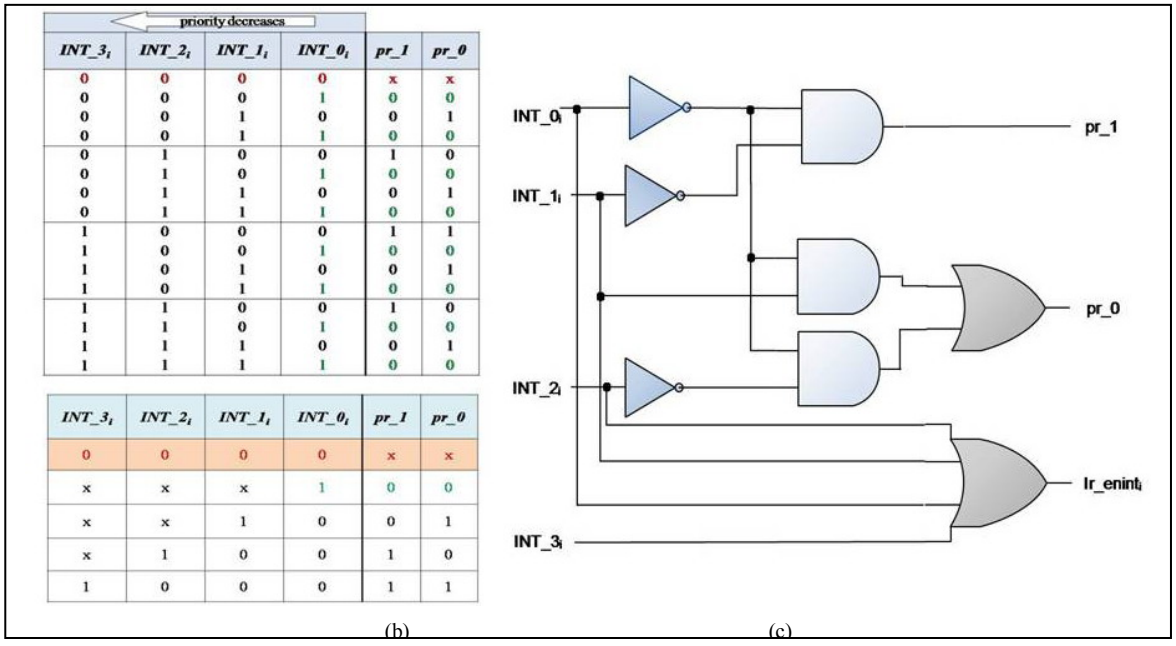

Fig. 9. Example of a figure caption. *(figure caption)*

The solution proposed in this paper is a somewhat based on the "interrupts as threads" concept presented in [9]. In [10], the authors present a software and hardware solutions to prevent the overload caused by the interrupts. The integrated model proposed in this paper can handle this overload through various scheduling techniques such as the use of sporadic servers [10]. Some RTOSs disable all external interrupts and treat them trought a polling mechanism on the timer interrupt[11]. In [12], the authors proposed a method in which interrupts are treated as threads. The proposal aims to increase the scalability of multiprocessor system architectures oriented to network server operating systems and the interrupt threads use specific priority levels. In [13], interrupts priorities can be dynamic (by reattaching to other task or by changing the priority of the task to which it is attached).

The proposed architecture uses a unified space of priorities for tasks and interrupts, and relies on the hardware activation mechanism of the tasks. In the architecture presented, as a continuation of research from [1] and [2], the interrupt handlers are treated as tasks. Usually, there are situations of priority inversion when high-priority tasks are suspended by the interrupts assigned to low priority tasks. The unification of the tasks and priorities in the same address space has the role to eliminate this disadvantage. The nHSE uses a unified space of priorities for interrupts and tasks, and a scheduling rule in which a high-priority task cannot be interrupted by interrupts assigned to lower priority tasks. This rule supports the need to ensure the meeting of the deadlines for the tasks that should provide a real-time response to external stimulus. The nHSE enables the activation or deactivation the interrupts. Interrupts follow the same execution procedure as tasks so that enabling or disabling their execution is performed using the same instructions that are addressed to tasks. In the *nMPRA*, each timer associated with a task can be configured to generate an interrupt when the time allocated to the task is nearing completion. A task can respond to an external event if the event is masked by blocking *wait* instruction. The *wait* instruction is very powerful because it allows synchronizing the execution while multiple events. Under software control, based on the $sCPU_i$ (task *i*) tasks, these events are treated and cleared.

The most modern RTOS have implemented several mechanisms for resource sharing, synchronization and communication between tasks, but they provide API functions that must be called individually. For example, you cannot expect an interrupt together with a semaphore and a message. The solution proposed in this paper, allows this operation mode.

IV. CONCLUSION

In this paper, we improve the CPU architecture presented in [3] by an innovative solution for prioritization of the interrupts attached to the same task. Unlike loop testing solution, the proposed solution provides a uniform response time for any interrupt. Furthermore, the proposed solution can provide static priorities for the interrupts. We can say that the presented solution contains a unitary interrupts management, and a hardware solution to attach the interrupts to the tasks of the RTOS implemented in hardware.

In the future, we will focus on the solution to create the priority encoder blocks depending on the number of attached interrupts and the possibility to upload direct to the CPU hardware the address of the interrupt handlers.

ACKNOWLEDGMENT

This paper was supported by the project "Sustainable performance in doctoral and post-doctoral research PERFORM–Contract no. POSDRU/159/1.5/S/138963", project co-funded from European Social Fund through Sectorial Operational Program Human Resources 2007-2013.

REFERENCES

[1] E. Dodiu, V.G. Gaitan, A. Graur, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – architecture description", IEEE 35'th Jubilee International Convention on Information and Communication Technology, Electronics and Microelectronics, Croatia, May 2012.

[2] E. Dodiu and V.G. Gaitan, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – concept and theory of operation", 2012 IEEE EIT International Conference on Electro-Information Technology, Indianapolis, IN, USA, 6-8 May 2012, ISBN: 978-1-4673-0818-2, ISSN: 2154-0373.

[3] V.G. Gaitan, N.C. Gaitan, I. Ungurean, "CPU Architecture based on a Hardware Scheduler and Independent Pipeline Registers", submitted in IEEE Transactions on VLSI System, 2014.

[4] L.E. Leyva-del-Foyo, and P. Mejia-Alvarez, "Custom Interrupt Management for Real-Time and Embedded System Kernels", in Proceedings of the 10th IEEE ECOOP Workshop on Exception Handling in Object Oriented Systems Development, 2005.

[5] Shi-Hai Zhu, "Hardware Implementation Based on FPGA of Interrupt Management in a Real-time Operating System", Information Technology Journal, 2013.

[6] B.C . Alecsa, "FPGA implementation of a matrix structure for integer division", Proceedings of the 3rd International Symposium on Electrical and Electronics Engineering, Galati, Romania, 2010.

[7] I. Liu, J. Reineke, E.A. Lee, "A PRET architecture supporting concurrent programs with composable timing properties", in Signals, Systems and Computers, 2010, Conference Record of the Fortz Fourth Asilomar Conference on (pp. 2111/2115), IEEE.

[8] M. Shahbazi, P. Poure, S. Saadate, M.R. Zolghadri, "Fault-Tolerant Five-Leg Converter Topology with FPGA-Based Reconfigurable Control", IEEE Transactions on, vol. 60, no. 6, June 2013.

[9] W. Hofer, D. Lohmann, F. Schele, W. Schroder-Preikschat, "SLOTH: Threads as Interrupts", 30th IEEE Real-Time Systems Symposium, ISBN: 978-0-7695-3875-4, 204-213, 2009.

[10] J. Mäki-Turja, G. Fohler , K. Sandström, "Towards Efficient Analysis of Interrupts in Real-Time Systems". 11th EUROMICRO Conference on Real-Time Systems, York, England, May 1999.

[11] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M. González Harbour, "A practitioner's handbook for real-time analysis", Kluwer Academic Publishers, 1993.

[12] K. Jeffay, D. L. Stone, "Accounting for Interrupt Handling Cost in Dynamic Priority Task Systems", Proc. of the IEEE Real-Time Systems Symposium, pp. 212-221, December 1993.

[13] L. Cheng-Min, "Nested interrupt analysis of low cost and high performance embedded systems using GSPN framework", IEICE Trans. Inform. Syst., E93-D: 2509-2519, 2010.