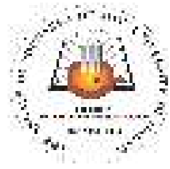


THE ANNALS OF "DUNAREA DE JOS" UNIVERSITY OF GALATI
FASCICLE IX. METALLURGY AND MATERIALS SCIENCE
N°. 1 - 2015, ISSN 1453 – 083X

DETAILED MICROCONTROLLER ARCHITECTURE BASED ON A HARDWARE SCHEDULER ENGINE AND INDEPENDENT PIPELINE REGISTERS

Lucian ANDRIES[1,2], Vasile Gheorghita GAITAN[1,2]
[1]Faculty of Electrical Engineering and Computer Science, Ștefan cel Mare University of Suceava
[2]Integrated Center for Research, Development and Innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for Fabrication and Control (MANSiD), Ștefan cel Mare University of Suceava
Suceava, Romania
e-mail: andries.lucian2002ro@gmail.com, gaitan@eed.usv.ro

ABSTRACT

In the world of real time operating systems, task switching, communication between threads and synchronization are implemented in software. Some of the mechanisms used may introduce big latencies in task recurrence, task jitter. This kind of problem, which is sporadic, may lead to system failure for safety-critical areas. This issue may occur in the real time systems that have really fast response time as requirements. For this particular example, the tasks are succeeding very fast, resulting in a lot of overhead because of the time spent in task switch. Our research has led us to the conclusion that a microcontroller architecture, based on a static hardware Scheduler and independent Pipeline Registers, will be capable of executing multiple tasks with approximately no delay between every task switch (5 machine cycles). The nMPRA (n Multi-Purpose Register Architecture) architecture, which consists of 2 sets of registers: local such as coprocessor 2 and global such as a peripheral on the slow bus, offers support for preemptive real time operating systems. Both architectures, nMPRA and nHSE (n Hardware Scheduler Engine), complement each other and take the real time operating system programming to a whole new level.

KEYWORDS: real time system, static hardware scheduler, microcontroller, pipeline processor

1. Introduction

The real time properties of an embedded system lie in the implementation of the operating system, task synchronization and communication between threads. The RTOS (real time operating systems), used, in a real fast system must be small, in order to have less overhead caused by the operating system for a fast code execution.

Currently, the real time operating system from the safety critical areas will not use a small operating system because a powerful microcontroller can be chosen in order to supplement the power needed for event serving. In article [1] a CPU architecture is described (central processing unit), that provides hardware support for real time systems and very good related work. This new architecture is composed of 2 different architectures:

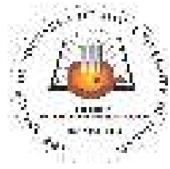
nMPRA (multiple pipeline registers architecture for n tasks): provides support for hardware synchronization between tasks and peripherals.

nHSE (hardware scheduler engine for n tasks): provides support for static and dynamic hardware scheduler for n tasks.

The combined architectures contribute to monitor resources and very fast events and they interrupt handling.

In [2] a MIPS processor was implemented in order to be modified for the new architecture. The MIPS architecture did not provide support for Coprocessor0 (COP0) or all the instructions that a MIPS32 processor can support. The only supported programming language was an assembler language, which was parsed by a tool chain written by us.

In article [1] the author points out that the Scheduler and the nHSE architecture can be included into a coprocessor or into the on-die implementation.



THE ANNALS OF "DUNAREA DE JOS" UNIVERSITY OF GALATI
FASCICLE IX. METALLURGY AND MATERIALS SCIENCE
N°. 1 - 2015, ISSN 1453 – 083X

The scheduler was implemented in the second manner while the nHSE architecture was implemented as a Coprocessor 2 (COP2) for the MIPS32 architecture.

First, in order to implement the Coprocessor 2, it was necessary to meet the following requirements:

A very tested functional MIPS32 architecture.

Support for coprocessor.

Support for a gcc (a compiler system produced by the GNU Project supporting various programming languages) tool chain and a high level programming language, which was considered as the most important part.

The project [3] which was thoroughly tested was used as a baseline for the new architecture. The project was created by Grant Ayers and funded by the eXtensible Utah Multicore (XUM) project at the University of Utah between 2010-2012.

The details of the classical 5-stage pipeline MIPS32 Release 1 architecture are:

- Harvard architecture with separate instruction and data ports.
- Full forwarding and hazard detection
- MIPS32 instruction set, including:
  - Atomic load linked.
  - Atomic store conditional.
  - Unaligned load and store.
- Complete Coprocessor 0 that allows ISA-compliant interrupts exceptions and user / kernel modes.

2. Detailed architecture of the microcontroller

The main differences between the processor located at [2] and the one implemented in [3] are the presence of the COP 0, support for all MIPS32 instructions, including the assembler instructions for COP 0 and the use of only one clock signal for the entire processor.

The MIPS32 Release 1 architecture is a project developed in Verilog using the Ide Xilinx ISE Design Suite 14.2, which was modified in order to support the architecture described in [1]. A multiplexer / demultimexer was introduced to manage the resources RAM, ROM, ALU, Control and Coprocessor 0, which are shared between the 5 threads and the static dual priority Scheduler (Figure 1).

The MIPS32 architecture is special because it left room for further implementations of new hardware modules. The MIPS32 architecture defines 4 coprocessors [4]:

- Coprocessor 0 (COP0): already implemented and supporting exceptions and virtual memory system.
- Coprocessor 1 (COP1): reserved for floating point custom implementation.
- Coprocessor 2 (COP2): available for user defined implementation.
- Coprocessor 3 (COP3): reserved for floating point module in Release 1 implementation of the MIPS64 architecture and for all Release 2 implementations of the architecture.

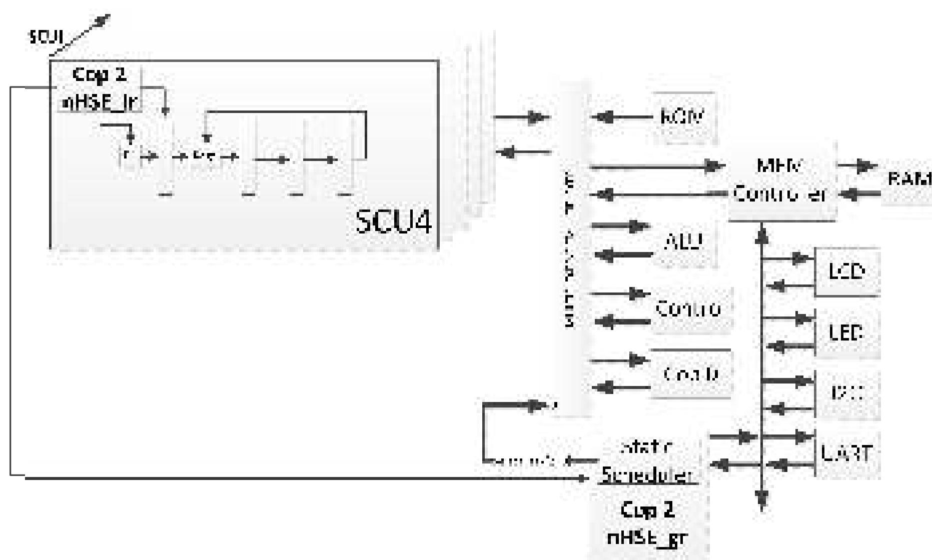


Fig. 1. *Controller with nHSE and nMPRA architecture*

The gcc tool chain also provides support for low level and high level programing language for the processor and assures us that we can use the 4 coprocessor in our implementation. The nHSE and nMPRA architecture will be tested using gcc, which is a widely used tool chain. In this way, we can create a benchmark between the old processor and the new architecture, using the same tool chain and programing language.

The nMPRA architecture was implemented to be COP2 compliant in order to benefit from the gcc tool chain. In Figure 1 is described, at a macro level, the cohesion between the nMPRA and nHSE architecture. SCPU0 (semi processor 0) will configure the Scheduler, which is a part of nHSE, to supervise the correct execution of the hardware tasks, because it has the highest priority and is the only hardware task that is allowed to configure the hardware Scheduler.

The nMPRA architecture contains:

- independent pipeline registers.
- a multiplexer and demultiplexer that share the resources that were not multiplicity.
- Local registers used for inter task synchronization.

The custom microcontroller can run 5 different tasks (5 ***SCPUi***) independently without the need of the nMPRA architecture. In this manner, the system can be considered as equivalent to the microcontroller with RTOS.

The nHSE architecture complements the nMPRA by adding an important part of an RTOS event, interrupts handling and inter task communication.

As described in article [1], the internal organization of the logic is divided into 2 different segments:

- The local segment consists of local registers that are visible only in the ***SCPUi*** (***nHSE_lr*** in Figure 1) and are located after the pipeline register ***Instruction Decode*** (***ID***). In this particular case, the access time to registers from ***nHSE_lr*** will be of 2 machine cycles because one machine cycle will be used to extract the instruction from ROM memory and the other machine cycle to decode and execute the COP2 instruction.
- The global segment consists of global registers from the external bus (***nHSE_gr*** in Figure 1). In this case, the access time will be of 5 machine cycles for our architecture because 4 steps will be driven through the pipeline registers until ***Memory Access*** (M) is reached and 1 machine cycle will be used to write or read from the ***nHSE_gr*** peripheral.

As a consequence of this architecture, the access to the local segment is faster than the access to the global segment.

The interaction between the static Scheduler and the nHSE resources is very tight, therefore the algorithm of the Scheduler, detailed in [2], was improved in order to support the most powerful instruction, wait (detailed in [1]). The nHSE_lr (Figure 2 a)) module will use TaskNeedReset wire to signal the Scheduler that the SCPUi must be reset because the watch-dog (register mrDEVi detailed in Chapter IV) has expired, and TaskDeepSleep to signal the Scheduler that the task is not blocked, but is only waiting for an external or internal event, in order not to be promoted to long task queue (***LTQ*** in [2]).

In order to have a better understanding of the Scheduler function, detailed in [2] and modified in Figure 3, we are going to repeat some relevant information.

Each ***SCPUi*** can belong to one of the following classes:

The class of active tasks (**AQ**) that are scheduled based on priorities in the Running State (***RS***) of the Scheduler.

The class of interrupted tasks (***ITQ***) that are scheduled based in priorities, only in the ***Idle State*** (***IS***) of the Scheduler.

The class of long execution tasks (***LTQ***) that are scheduled based on a ***Round Robin*** (***RR***) algorithm only in the Idle State (***IS***) of the Scheduler.

The wait instruction allows for the synchronization with several events that are located into the ***crTRi*** (Table 1) and ***crEVi*** registers, without the need of the software intervention.

Table 1. *crTRi (Control registers)*

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

lr_enTi: validates/inhibit generated timer event.

lr_enWDi: validates / inhibits the events generated by the watchdog.

lr_enD1i: validates / inhibits the events generated by deadline 1.

lr_enD2i: validates / inhibits the events generated by deadline 2.

lr_enInti: validates / inhibits the events generated by interrupts.

lr_enMutexi: validates / inhibits the events generated by mutex.

lr_enSyni: validates / inhibits the events generated by timing events.

lr_run_sCPUi: validates / inhibits the program execution SCPU.

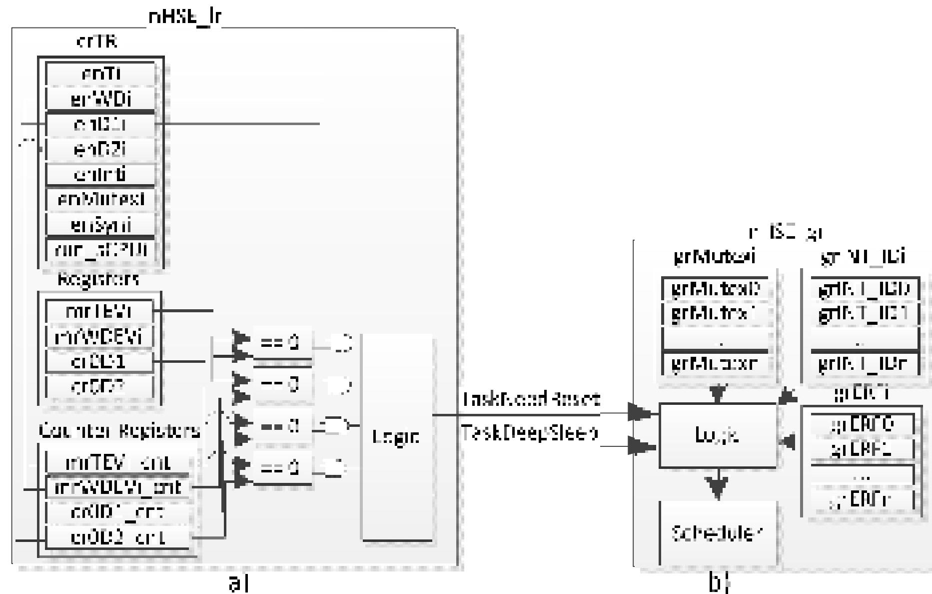


Fig. 2. *nHSE_lr and nHSE_gr internal architecture*

Table 2. *crEVi (Events Register)*

31..	7	6	5	4	3	2	1	0
..	lr_run_sCPUi	SynEvi	MutexEvi	IntEvi	D2EVi	D1EVi	WDEvi	TEvi
	rw	rw	rw	rw	rw	rw	rw	rw

TEvi: the event generated by the timer has occurred.

lr_enWDi: the event generated by the watchdog has occurred.

lr_enD1i: the event generated by deadline 1 has occurred.

lr_enD2i: the event generated by deadline 2 has occurred.

lr_enInti: the event generated by interrupts has occurred.

lr_enMutexi: the event generated by the mutex has occurred.

lr_enSyni: the event generated by timing events has occurred.

lr_run_sCPUi: the copy of the bit lr_run_sCPUi from register crTRi.

The current assembler instruction, which is decoded by ***nHSE_lr*** (Figure 2, a)) module, is relatively unique because one atomic instruction will stop the current ***SCPUi*** from execution, equivalent with entering a low power mode, and will alert the Scheduler not to promote the current task to ***LTQ*** through the wire ***TaskDeepSleep*** (Figure 2).

The static scheduler algorithm, reused from [2], modified and detailed in Figure 3, supports a self-sustaining state (***Waiting State*** from Figure 3) that my last as long as the task needs it. This feature assures us that the current modified architecture is not rigid; on the contrary, we could say that it can support a large number of tasks.

The Scheduler is incorporated into ***nHSE_gr*** (Figure 1) because it must have access to the global events, presented in ***Chapter IV***, in order to stop the active ***SCPUi*** and schedule the right ***SCPUi*** to serve the event or just schedule the correct ***SCPUi***.

The number of registers that are used for inter task synchronization, such as mutex or message passing, must be standardized in order to serve all the

tasks. For example, the minimum number of registers for mutexes (***grMutexi***) and message passing (***grERFi***) must be equal to the number of tasks to the power of two. This allows us to use a mutex or send a message from one to all the others.

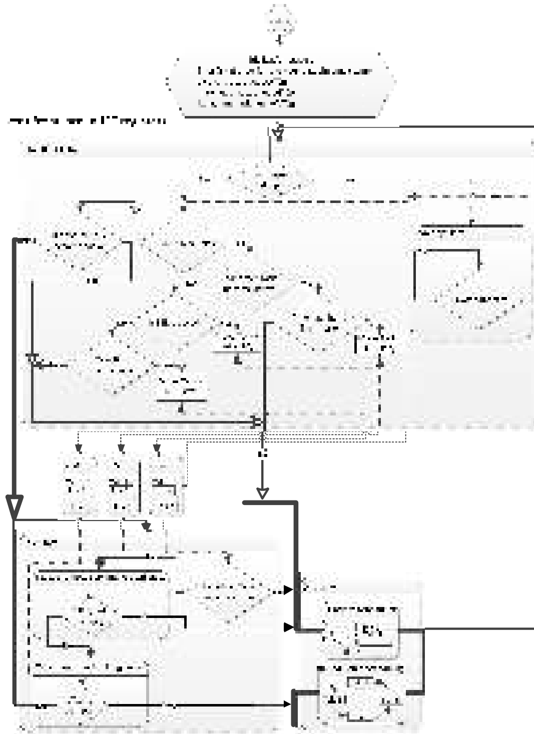


Fig. 3. *The flowchart of the scheduler with a dual priority algorithm*

3. Local events

The internal architecture of ***nHSE_lr*** block is described in Figure 2, a).

*Watch dog timer(***mrWDEVi***)*

If the watchdog is configured and the timer expires, the ***SCPUi*** will be reset. The timer of the watch dog will be reset each time the task has ended successfully.

*Alarms(**mrD1EVi**, **mrD2EVi**)*

Each ***SCPUi*** has 2 alarms that will signal, via ***lr_enD1i*** and ***lr_enD2i***, that the configured time has passed. The alarms may be used to find out how long the execution of the code lasted.

Every time the task begins again to execute the code, the local registers (Figure 2, a)) ***mrTEVi_cnt***, ***mrWDEVi_cnt***, ***cr0D1_cnt*** and ***cr0D2_cnt*** will be initialized with the values defined in the related registers. After the initialization process, the registers will start decrementing the defined values. When the counters are equal to 0, a flag will be set and further action will be taken, depending on each register functionality.

*Timers(**mrTEVi**)*

Can be used as a general purpose timer to wait for an event or just for synchronization with other tasks or events. The wait instruction can be used to generate a fixed recurrence of a task when an input is scanned periodically. Thus, the processor will consume less power.

4. Global events

The internal architecture of ***nHSE_gr*** block is described in Figure 2 b).

Mutexes

Table 3. *grMutexi (task ID for interrupt attached register)*

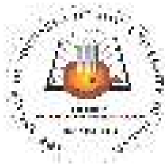
31	30..4	4	3	2	1	0
TaskID31	..	TaskID4	TaskID3	TaskID2	TaskID1	TaskID0
Mutexi	0	rw	rw	rw	rw	rw

TaskID = the task id that acquired the mutex.

Mutexi = 1 means the mutex is taken.

Acquire / release a mutex (Table 3): The operation is atomic (write to ***nHSE_gr***) because the data path of the processor is not used. Instead it is used a specific asm instruction from COP2 (Figure 2 a)) (***mtc2***). The ***nHSE_lr*** is going to decode the instruction and write the relevant data to ***nHSE_gr*** to acquire the mutex. A mutex can be released only by the task that acquired it.

Read the status of a mutex: the operation uses the data path of the SCPUi, because the MIPS architecture can perform only operations based on registers. Therefore, a value from RAM or peripheral must be stored in one of the 32 registers from SCPUi in order to be used in computation.



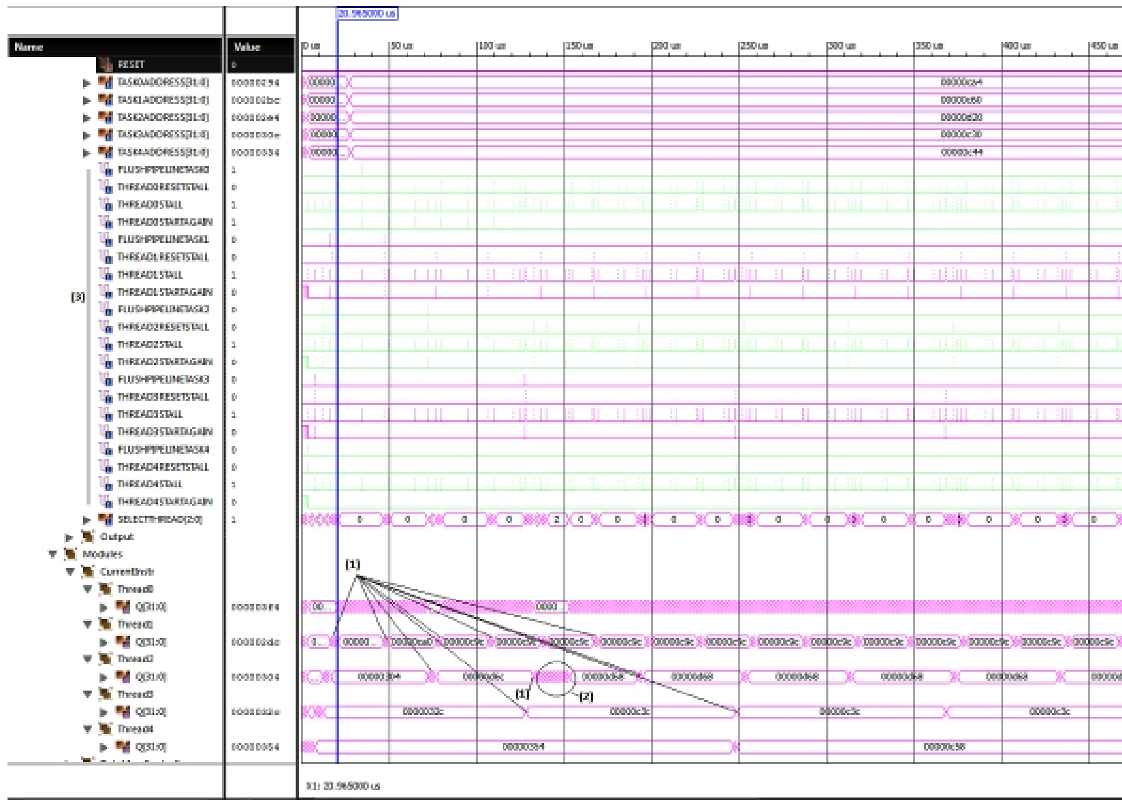


Fig. 4. *Scheduling waveform of 5 hardware tasks (capture from Xilinx ISE Simulator)*

Interrupts

Table 4. *grINT_Idi (Task ID for interrupt attach register)*

31	30..3	2	1	0
	…	TaskId 2	TaskId 1	TaskId0
0	0	rw	rw	rw

The register grINT_IDi selects the id of the SCPUi that will execute the interrupt routine code (Table 4). A simple explanation is that a SCPU4 can only execute the interrupt service routine of a GPIO (General Purpose Input Output) pin while the SCPU3 can only execute the interrupt service routine of an exception from the communication peripheral.

Events

Table 5. *grERFi (event global register)*

	Nj-1	..	0	nj-1	..	0	k-1	..	0
0/1	sIDnj-1	..	sID0	dIDnj-1	..	dID0	mess	..	mess0
event	Source task id			Destination source task			message		
rw	rw		rw	rw		rw	rw		rw

Event = 1 means the event is active.

The task will be awakened by the Scheduler when the ***wait*** assembler instruction is used and a message is received. The data path of the SCPUi will be used to access data from the register (Table 5).

5. Simulation

In Figure 4 (1) is visible the PC (Program Counter) of each ***SCPUi*** (***Modules\CurentInstr***), that whenever is changing the current of the ***PC***, the line will be changed. By looking at this pattern we can see that the hardware tasks are scheduled periodically at the same recurrence.

In Figure 4 (2) the ***SCPU2*** hardware task (***Modules\CurentInstr\Thread2\Q***) does not respect the pattern because it is executing more codes, for the first time, than the other ***SCPUi***. The code for all ***SCPUi*** was written in this manner on purpose, to show that the hardware tasks are scheduled regularly at the same recurrence. In Figure 4 (3), the hardware signals are used to perform a task switch of a ***SCPUi***. A more detailed explanation about the task switch is provided in [2]. The current response time of the nHSE architecture, using Verilog simulation is the following:

nHSE_lr – 1 machine cycle for internal handling, where the atomic instructions are executed in 2 machine cycles. One machine cycle to extract the instruction and another machine cycle to execute de instruction.

nHSE_gr – 1 machine cycle for internal handling.

Scheduler – 1 machine cycle to schedule a task (response time) and 5 machine cycles to perform a task switch. The process of a task switch is described in [2].

The above response time is obtained using simulation and can be different from the case where the design is synthetized, because it depends on the developer that is designing the RTL (register-transfer level).

The current proposed solution is completely simulated using ISE Design Suite 14.2 and ISE Simulator, proving that the functionality, proposed in [1], is working and can be implemented in hardware. A problem that must be handled in the future is that different authors have made some significant improvements, such as improved response to interruptions [5, 6] and events [7, 8]. Therefore, the nHSE features must be extended and new registers for configuration must be added.

6. Conclusions

In this paper we presented a possible implementation of the nHSE architecture as a MIPS coprocessor, Coprocessor2 (COP2). We outline, for the first time, the nHSE registers and some of their functionalities for the local registers (coprocessor 2) and global registers (such as the peripheral on the slow bus).

The solution has the great advantage of being able to take advantage of all gcc tool chain support, that can use a high level programming language to generate machine code for the modified MIPS processor architecture, was considered the most important part.

This new architecture has a response time for the nMPRA architecture of 3 machine cycles for the local registers (***nHSE_lr***), 1 machine cycle for global registers (***nHSE_gr***) and 6 machine cycles for the Scheduler in order to respond and make a task switch.

The solution has been implemented using Verilog, as a hardware description language, and ISE Simulator for simulation purpose.

We can say that the overall response time of this architecture is less than the execution of a load instruction.

Acknowledgement

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.

This work was partially supported from the project Integrated Center for research, development and innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for fabrication and control, Contract No. 671/09.04.2015, Sectoral Operational Program for Increase of the Economic Competitiveness co-funded from the European Regional Development Fund.

References

[1]. Gaitan V. G., Gaitan N. C., Ungurean I., *CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. PP, no. 99, p. 1, ISSN: 1063-8210, doi: 10.1109/TVLSI.2014.2346542.

[2]. Andries L., Gaitan G., *Dual priority scheduling algorithm used in the nMPRA microcontrollers*, System Theory, Control and Computing (ICSTCC), 2014 18th International Conference, p. 43-47, 17-19 Oct. 2014, doi: 10.1109/ICSTCC.2014.6982388.

[3]. ***, https://github.com/grantea/mips32r1_xum.

[4]. ***, *MIPS Technologies, Inc. 955 East Arques Avenue Sunnyvale, CA 94085-4521*, Document Number: MD00082 Revision 3.02, March 21, 2011.

[5]. Gaitan N. C., Gaitan V. G., Moisuc E.-E. C., *Improving interrupt handling in the nMPRA*, Development and Application Systems (DAS), 2014, p. 11-15, 15-17 May 2014 doi: 10.1109/DAAS.2014.6842419.

[6]. Gaitan N. C., Andries L., *Using Dual Priority scheduling to improve the resource utilization in the nMPRA microcontrollers*, Development and Application Systems (DAS), 2014, p. 73-78, 15-17 May 2014, doi: 10.1109/DAAS.2014.6842431.

[7]. Moisuc E.-E. C., Larionescu A.-B., Gaitan V. G., *Hardware event treating in nMPRA*, Development and Application Systems (DAS), 2014, p. 66-69, 15-17 May 2014, doi: 10.1109/DAAS.2014.6842429.

[8]. Moisuc E.-E. C., Larionescu A.-B., Ungurean I., *Hardware event handling in the hardware real-time operating systems*, System Theory, Control and Computing (ICSTCC), 2014, p. 54-58, 17-19 Oct. 2014, doi:10.1109/ICSTCC.2014.6982390.