World Academy of Science, Engineering and Technology
International Journal of Electrical and Computer Engineering
Vol:11, No:9, 2017

CPU Architecture Based on Static Hardware Scheduler Engine and Multiple Pipeline Registers

Ionel Zagan, Vasile Gheorghita Gaitan

***Abstract*—**The development of CPUs and of real-time systems based on them made it possible to use time at increasingly low resolutions. Together with the scheduling methods and algorithms, time organizing has been improved so as to respond positively to the need for optimization and to the way in which the CPU is used. This presentation contains both a detailed theoretical description and the results obtained from research on improving the performances of the nMPRA (Multi Pipeline Register Architecture) processor by implementing specific functions in hardware. The proposed CPU architecture has been developed, simulated and validated by using the FPGA Virtex-7 circuit, via a SoC project. Although the nMPRA processor hardware structure with five pipeline stages is very complex, the present paper presents and analyzes the tests dedicated to the implementation of the CPU and of the memory on-chip for instructions and data. In order to practically implement and test the entire SoC project, various tests have been performed. These tests have been performed in order to verify the drivers for peripherals and the boot module named Bootloader.

***Keywords*—**Hardware scheduler, nMPRA processor, real-time systems, scheduling methods.

I. Introduction

IN society today, the real-time systems (RTS) used in the automotive industry, automation, nuclear power engineering, in the field of aerospace, or in medical systems, and not only, represent a very important and essential category, where the processor is a particularly important item. In most systems deployed in the areas mentioned above, processors do not only ease human work in the production or development process, but also represent a central element favoring an increase in the quality of products. This enhances competitiveness in industry, and the importance of the RTS increases as they are used also for eliminating the occurrence of material damage [1].

The real-time kernel is the heart of any RTS, directly connected to the hardware of the physical environment. Usually, the kernel performs the following actions: Process management; interrupts management; process synchronization.

Process management is the main service provided by the real-time kernel. This involves the implementation of certain functions, as creating and terminating processes, scheduling jobs, other scheduling operations, context switching and various relative activities.

The interrupts handler is that part of the executive designed to guarantee the management of interrupts which can be generated by peripheral devices such as keyboard, analog-digital converters, or even certain sensors [1]. This mechanism implies the execution of a driver for each interrupt, in order to transfer data from the peripheral device towards or from the main memory. A different functionality of the kernel is that of supplying the basic mechanisms for process synchronization and communication. In classical operating systems, these mechanisms are represented by semaphores that enable the mutual exclusion of tasks for shared resources. However, the semaphores can cause priority inversion, leading to the unlimited blocking of tasks execution. As a consequence, in order to guarantee predictability, the hard real-time operating system (RTOS) must provide special semaphores supporting access protocols to shared resources, such as Priority Inheritance, Priority Ceiling, or Stack Resource Policy; this way, priority inversion should be avoided.

Of the other activities of the kernel, we can mention high-level services of the operating system, and the initializing of internal data structures, such as Task Control Block (TCB), queues, tables and semaphores [2].

This paper describes the implementation of nMPRA processor [3], [4] using the FPGA Virtex-7 circuit, presenting the experimental model of the static real-time scheduler implemented in hardware. For this to be obtained, the field-programmable gate array (FPGA) devices [5], available today at acceptable prices [6], [7], represents a hardware support for the development of nMPRA processor.

This paper is structured as follows: After a short introduction in Section I, Section II briefly describes the nMPRA architecture. Section III presents the experimental data obtained from the validation of the instructions dedicated to the static scheduler, and Section IV explains the access to peripherals and to the boot procedure; the last section of the paper presents the conclusions and further research directions.

II. NMPRA Architecture and Hardware Support

In order to obtain a competitive processor, we focus on the nMPRA architecture, as a RTOS developed in hardware, based on a Hardware Scheduler Engine (nHSE). The architecture of the nMPRA processor is based on a five-stage assembly line, enabling the simultaneous execution of up to five instructions on different stages [8], [9].

The original design is based on a traditional MIPS architecture that was specially modified to support instructions dedicated to the hardware scheduler, part of the CPU itself.

Ionel Zagan and Vasile Gheorghita Gaitan are with the Stefan cel Mare University of Suceava, Suceava, Romania and 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 (e-mail: zagan@eed.usv.ro, vgaitan@eed.usm.ro).

The MIPS32 architecture is a superset of previous MIPS I and MIPS II; these architectures incorporate powerful new instructions, especially for embedded applications, as well as for the management of the control mechanism and of memory, for the privileged mode, previously found only in R4000 and R5000; R4000 is one of the first 64-bits processors and the first implementation of the MIPS III. By incorporating these powerful new features, standardizing the instructions of the privileged mode, and maintaining compatibility with the architecture of older CPU's instruction set, the MIPS32 architecture provides a solid and high performance foundation for all future developments on these processors [10].

As it can be seen in Fig. 1, the authors use a register file and a set of four pipeline registers for each task, in order to hold the individual running state information. All the tasks running on the CPU use the same data path, control unit, ALU, Hazard Detection Unit and Forward Unit [11]. So, an instance of the CPU will be called semi processor (sCPUi for task i).

The implementation is based on the project described in [12], a 32-bit MIPS processor which aims for conformance with the MIPS32 Release 1 ISA. The project has been implemented using the VC707 Evaluation Kit [13] produced by Xilinx and Vivado 2015.4 design environment and the source code has been written in Verilog HDL.

The nHSE is a finite state machine which has inputs for events, such as interrupts, deadline, watchdog timers, timers, mutexes, messages, and self-support execution [14], [15]. The static scheduler is task-oriented. The priority of each sCPUi is *i*, the same with the ID of the sCPUi; this means that priorities are constant during the execution. Based on the remapping of the task contexts, this implementation allows a very fast task switching operation. In other words, the nMPRA architecture replaces the classical stack-saving methods with a remapping technique, allowing us to execute a new task in an average of one clock cycle.

The paper also shortly describes the SoC project that integrates the nMPRA processor, the dual-port memory, the drivers for UART communication, the input/output components (LCD, DIP selectors and LEDs), as well as the debug component and also the way in which the Boot protocol is implemented and interacts with the CPU. A detailed comparison between the nMPRA architecture and other similar projects can be found in [4].

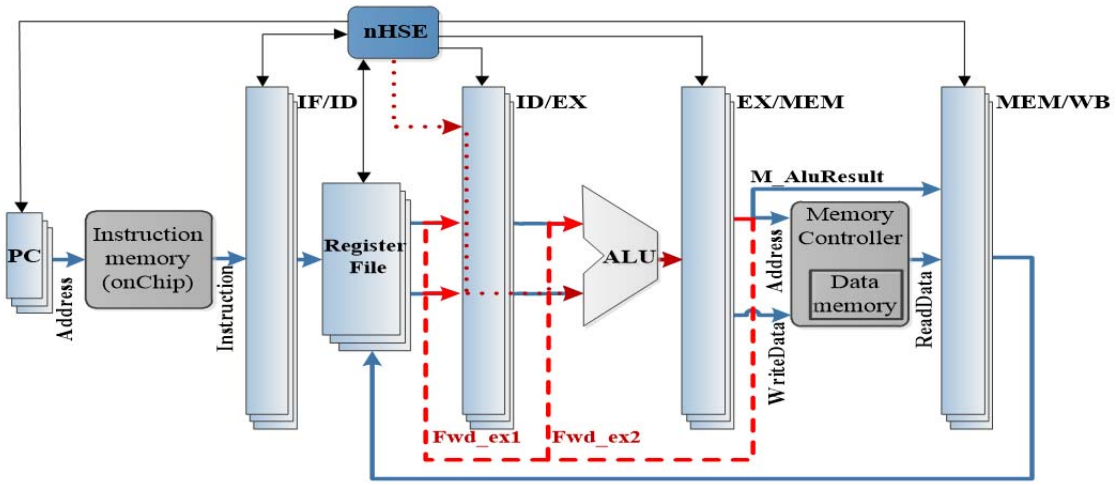

Fig. 1 Replication of resources of the nMPRA architecture and redirecting data in the hazard situations presented in Fig. 3. nHSE: Hardware Scheduler Engine, PC: program counter, IF: instruction fetch, ID: instruction decode, EX: execute, MEM: memory, WB: write back stage

III. The Experimental Results Obtained During the Testing Stage of Instructions Dedicated to the Static NHSE Scheduler

The implementation of new additional instructions necessary for the functioning of the static nHSE unit represents the novelty brought by the present paper. Thus, data transfer between nHSE and COP0 is achieved by using four instructions implemented at the level of coprocessor 2, namely CFC2 (Copy control register From COP2), CTC2 (Copy control register To COP2), MFC2 (Move monitoring register From COP2) and MTC2 (Move monitoring register To COP2). This paper is not aimed at describing in detail the instructions and the registers of the nHSE architecture, but at implementing and testing these instructions at the level of the static nHSE scheduler; it also presents the solution for the occurrence of hazard situations. Besides these instructions, the *wait* instruction is added; this instruction holds a special status, because it allows the modification of the active sCPUi's crTRi registry, performing at the same time the context switching operation, if the sCPUi executing this instruction has no other active events. The syntax of these instructions depends on the implementation, and is described in a broader paper focusing on the specifications of the nMPRA architecture.

The OpCode=6'b110010 field is the same for all instructions listed above; the interpretation and the format of the following fields Rs, Rt and Offset or Impl is specific to the implementation of Coprocessor 2.

Fig. 2 shows the clock signal of the nMPRA processor, the mrCntRuni[0:3] (the monitoring register accessed only by the sCPU0, and possibly by the monitored sCPUi, enables the reading of the operation cycle of each sCPUi), the crTRi[0:3][31:0] (is a control register with the role of validating (1) or inhibiting (0) an event), the mrTEVi[0:3][31:0] (the monitoring register memorizes the value with which the timer for sCPUi is reloaded), the grINT_IDi[0:3][31:0] global register (the register that selects the ID of the task to which the interrupt has been attached); these are only some of the nHSE scheduler's registers. The nHSE_Task_Select[3:0] selector can be observed in the lowest part of the simulation, together with the validation signal of the nHSE_EN_sCPUi semiprocessors; this selector is necessary for the context switching operation.

In the following part of the paper, we will exemplify the execution of the 0x48020000 instruction dedicated to the nHSE, even when data hazard occurs. The 0x48020000 instruction type MFC2 copies the mrTEVi[sCPU0] monitoring register from nHSE to the RF_registers[2] register. As an example, the 0x20010011 and 0x24420001 MIPS instructions have been considered; the 0x20010011 instruction updates the RF_registers[1] register with the immediate value of 0x00000011, and the 0x24420001 instruction adds the immediate value 0x00000001 to the RF_registers[2] register; the result is stored in RF_registers[2].

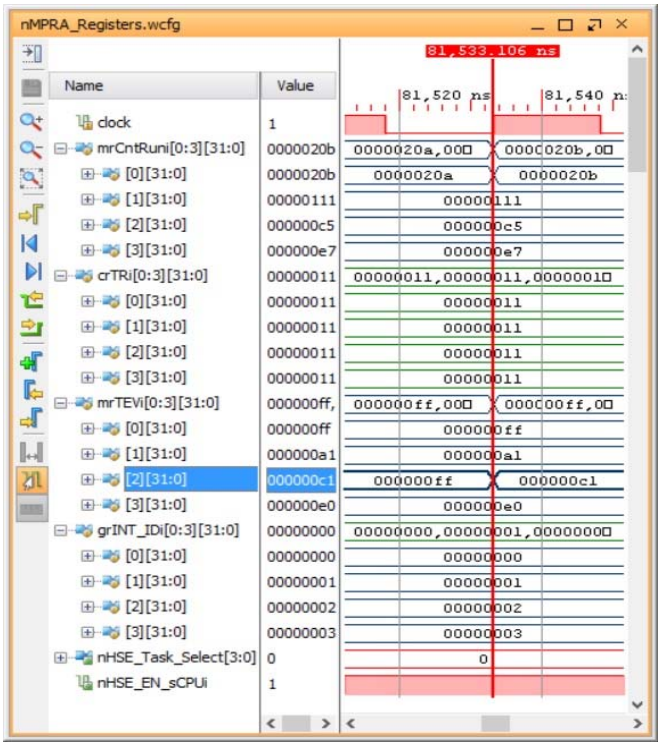

Fig. 2 Command, control and status registers that influence the nHSE directly or indirectly

To better exemplify the waveforms in Fig. 3, arrows indicating the instructions extracted from the program memory and their effect have been introduced. The arrows pointing downwards indicate the transition between the ID, EX, MEM and WB pipeline stages, whereas the two arrows pointing upwards, Fwd_ex1 and Fwd_ex2, indicate the forwarding of data. Time moment T1 indicates the 0x48020000 instruction which transfers the 0x000000FF value from the mrTEVi[sCPU0] monitoring register to the RF_registers[2] register. The execution of the 0x24420001(R[rt]=R[rs]+ SiggnExtImm, where Opcode=addiu, rt=2, rs=2 and SiggnExtImm=1) MIPS instruction is marked by time moment T2, and the result is sent to the M_ALUResult_reg[0][31:0] and WB_ALUResult_reg [0][31:0] multiplied registers via the M_ALUResult and WB_ALUResult signals. In case of decoding the 0x24420001 instruction, at the time moment marked by marker C1, the ID_RsFwd_Sel[1:0] and ID_RtFwd_Sel[1:0] control signals select the operands needed in the arithmetic operation, even in the presence of a Fwd_ex1 hazard situation. At the end of the simulation, the 0x48C10000 (*wait* RF_registers[1]) was tested; this instruction copies the value contained in the RF_registers[1] register of COP0 in the crTRi[sCPU0] control register of nHSE, and, at the same time, performs the context switching operation between sCPU0 and sCPU3. Fig. 1 shows the block diagram for the execution of instruction 0x24420001 in two different contexts; the result is two cases of hazard, corresponding to T2 and T3 time moments, marked in Fig. 3. Thus, depending on the occurred situation, the redirecting of data in the decoding or execution stage is performed using the Fwd_ex1 and Fwd_ex2 paths. For this, it was necessary to modify the control unit, the unit for the detection of hazards and forwarding of data, because the redirected register comes from coprocessor 2. Fig. 3 represents the situation when there are two hazard situations on the assembly line; time moment T4 represents the moment in which context switching is performed; the data stored in the pipeline registers are saved during the transition from sCPU0 to sCPU3. This switch takes place under the strict command of the nHSE static scheduler, through the nHSE_Task_Select[3:0] and nHSE_EN_sCPUi nHSE signals, the time needed for switching contexts is no more than one clock cycle.

The simulation illustrated in Fig. 3 shows the propagation of the control signal and the redirection of data from MEM and WB pipeline stages, in case a register is copied from the nHSE (coprocessor 2) to register file (coprocessor 0).

The simulation illustrates the signals generated by the Hazard_Detection module to flag an occurred hazard situation [16]; the data redirecting unit selects the source of the operands in the case of the encountered hazard, through the ID_RsFwdSel, ID_RtFwdSel, EX_RsFwdSel and EX_RtFwdSel lines. It can be seen that the EX_ALUResult register contains the result of the performed operation, while the EX_EXC_Ov register may indicate an overflow-type exception. Furthermore, the variation of the EX_AluSrcImm selection signal for the selection multiplexor of an Immediate-type operand can also be seen. As it is shown by the arrow pointing upwards, the redirecting of data takes place from the MEM stage to the ID_ReadData1_End register in the ID stage.

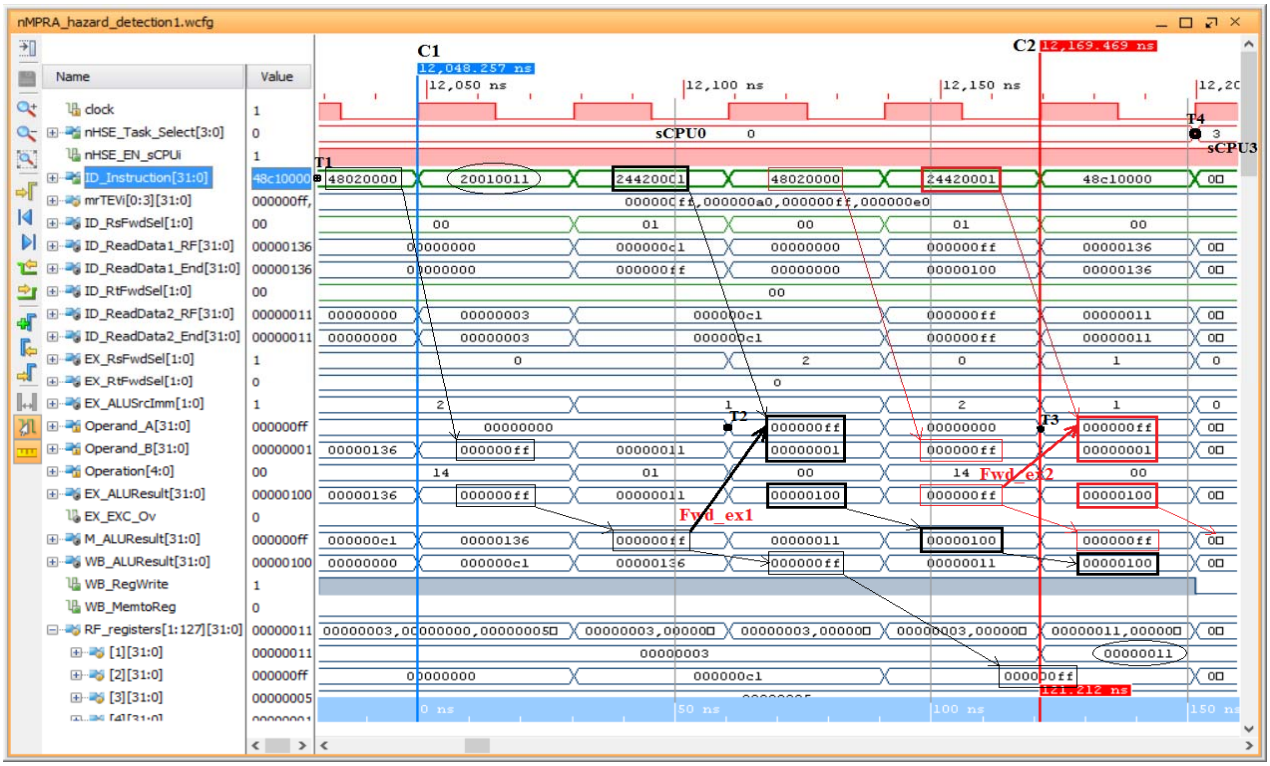

Fig. 3 Data redirection in the case of hazard situations, when instructions of the nHSE scheduler are executed on the assembly line. *clock* - nMPRA clock; *nHSE_EN_sCPUi* - nHSE enable signal; *nHSE_Task _Select[3:0]* - nHSE task selector; *ID_Instruction[31:0]* - wire type instruction; *ID_Instruction_reg[0:3][31:0]* - reg type sCPUi instruction; *RF_registers[1:127][31:0]* – Register File

At the next clock cycle, the 0x000000ff data is copied in the Operand_A register of the execute stage, exemplifying the Fwd_ex1 case shown in the block scheme from Fig. 1. The ID_ReadData1_RF and ID_ReadData2_RF signals represent the data read from the register file, whereas ID_ReadData1_End and ID_ReadData2_End send the data selected by the ID_RsFwdSel[1:0] and ID_RtFwdSel[1:0] signals.

At time moment T2, the 0x48020000 instruction is again executed, thus generating the hazard situation labeled as Fwd_ex2 in Fig. 1. At time moment T3, when executing the 0x24420001 instruction, the data redirection occurs from the memory pipeline stage to the execute stage, through the Ex_RsFwdSel[1:0] signals.

The M_ALUResult data, representing the result of the operation performed in the previous pipeline stage, and the content of the RF_registers[1:127][31:0] register file are shown in Fig. 3. Moreover, the writing of the 0x00000011 value in the EX/MEM pipeline register and later in the RF_Registers[1], at the time moment marked by marker C2 (as it is showed by the simulator), is represented in the same Fig. 3. The waveforms obtained for the WB pipeline stage can be observed; it is also important to note the correspondence between the executed instructions and the data written in the WB_ALU_Result, and afterwards in the register file. The WB_MemtoReg signal represents the one bit control line required for the selection of data sent in the register file, through the WBMemtoReg_Mux multiplexer. The WB_RegWrite control signal writes WB_ALU_Result in the register file on the positive front of the clock signal.

IV. Testing the Access to Peripherals and the Boot Procedure

As for the boot procedure, the uart_bootloader module represents a UART compatible RS-232 driver, implemented in Verilog HDL. The VC707 evaluation kit uses Standard Microsystems Corporation USB3320 USB 2.0 ULPI Transceiver (U8) in order to support the USB connection to the host computer [13]. In order to allow connections to a PC using the USB port, the development kit contain a Silicon Labs CP2103GM USB-to-UART (U44) bridge device, the USB cable being supplied in VC707 board. The FPGA circuit supports USB-to-UART bridge using four lines (Transmit (TX), Receive (RX), Request to Send (RTS) and Clear to Send (CTS)); for this purpose, Silicon Labs provide the Virtual COM Port drivers (VCP). These drivers enable the bridge circuit to be accessed as a COM port within the BootLoader application implemented and run on the host PC [12]. Through this PC application and through the uart_bootloader module, the program memory implemented in the FPGA will be rewritten, thus allowing the change of instructions executed by the nMPRA processor. The UART is of general use, able of receiving and sending data with a predetermined transfer rate. The UART communication, type 8N1, uses 8 bits of data, one stop bit, with no parity, and only RxD and TxD signals. The module uses two buffer registers of 256 bytes each, one for sending, and the other for receiving data. At reset, the

bootloader is enabled by default. When the bootloader is enabled, the memory bus will not detect any input data. To configure the UART communication in the general-purpose module, the program must first initiate writing command towards the UART on the data memory bus with bit 8 set. This will disable the boot protocol until the UART is reset again, thus enabling normal booting. It should be noted that there are only five seconds after reset during which the bootloader is active [12]. Afterwards, the software status determines the UART's mode of operation.

Fig. 4 shows a screen-shot made with the ChipScope analyzer, representing the content of the registers used for receiving data by the UART driver using the XUM communication protocol [12]. It can be seen how the line UART_Interrupt signals the transmission of the 0x58 byte saved in the temporary register uart_data_in[7:0]. The following lines of code set the reception-transmission pins in the constraint file: set_property PACKAGE_PIN AU33 [get_ports UART_Rx], set_property PACKAGE_PIN AU36 [get_ports UART_Tx].

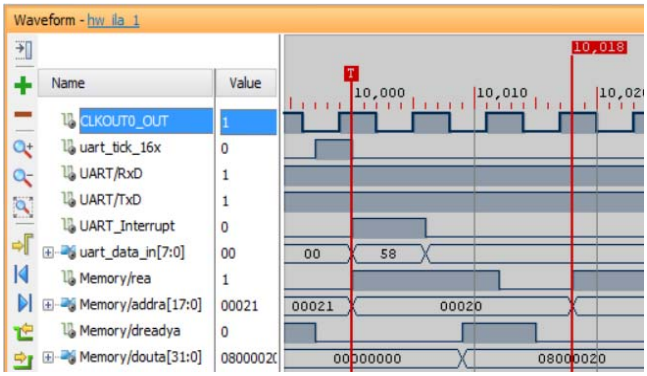

Fig. 4 Verifying the data received on UART

To test the access to an output port connected to a LED located on the Virtex-7 development board, it was also used the mapping of outputs in the address space of the data memory; the program performed a simple switching of a pin set to the .xdc constraints file. The LED[0] output is configured by setting the following properties: PACKAGE_PIN AM39, IOSTANDARD LVCMOS18, DRIVE 12 and SLEW SLOW.

V. Conclusion and Future Work

The modular implementation of the processor, the interface with memory separate from the processor, the individual design of drivers for UART and I/O, the port mapping in memory address space, as well as legibility and readability of the code, all are parts of the SoC project presented in this paper. Moreover, the flexibility of the project enables it to be easily tested and improved in subsequent research activities.

As future work, we aim to present the summary of the power consumption for the proposed nMPRA processor with 4 sCPUi, 8 sCPUi and 16 sCPUi, and the experimental results obtained from including in hardware the inter-task synchronization and communication mechanisms.

Acknowledgment

This paper was supported by the project "Development and integration of a mobile tele-electrocardiograph in the GreenCARDIO© system for patients monitoring and diagnosis - m-GreenCARDIO", Contract no. BG58/30.09. 2016, PNCDI III, Bridge Grant 2016, using the infrastructure from the project "Integrated Center for research, development and innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for fabrication and control", Contract No. 671/09.04.2015, Sectoral Operational Program for Increase of the Economic Competitiveness co-funded from the European Regional Development Fund.

References

[1] G. C. Buttazzo, "Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications," Third edition, Springer, 2011, ISBN: 978-1-4614-0675-4.
[2] W. Stallings, "Computer Organization and Architecture," 10th Edition, 2015, ISBN: 978-0134101613.
[3] E. Dodiu and 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, IN, USA, pp. 1-5, May 2012.
[4] V. G. Gaitan, N. C. Gaitan, and I. Ungurean, "CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 9, pp. 1661-1674, Sept. 2015.
[5] J. Shawash, and 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.
[6] M. Shahbazi, P. Poure, S. Saadate, and 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.
[7] T. T. Phuong, K. Ohishi, Y. Yokokura, and C. Mitsantisuk, "FPGA-based high-performance force control system with friction-free and noisefree force observation," IEEE Trans. Ind. Electron., vol. 61, no. 2, pp. 994–1008, Feb. 2014.
[8] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design, Revised Fourth Edition: The Hardware-Software Interface," Fourth Edition, 2011, ISBN: 978-0-12-374750-1.
[9] I. Zagan, "Improving the performance of CPU architectures by reducing the Operating System overhead," in the *3rd IEEE Workshop on Advances in Information, Electronic and Electrical Engineering AIEEE'2015*, pp. 1-6, 13-14 Nov. 2015, Riga, Latvia.
[10] "MIPS® Architecture for Programmers Volume I-A: Introduction to the MIPS32® Architecture," Revision 3.02, Mar. 2011, Available: https://courses.engr.illinois.edu/cs426/Resources/MIPS32INT-AFP-03.02.pdf (Accessed: May 2016).
[11] I. Zagan and V. G. Gaitan, "Schedulability Analysis of nMPRA Processor based on Multithreaded Execution," in *13rt International Conference on Development and Application Systems*, Suceava, Romania, pp. 130-134, May 19-21, 2016.
[12] http://opencores.org/project,mips32r1 (Accessed: Sept. 2015).
[13] www.xilinx.com/support/documentation/boards_and.../ug885_VC707_Eval_Bd.pdf (Accessed: Aug. 2016).
[14] E. E Moisuc, A. B. Larionescu, and V. G. Gaitan, "Hardware Event Treating in nMPRA," in 12rt International Conference on Development and Application Systems, Suceava, Romania, pp. 66-69, 15–17 May, 2014.
[15] I. Zagan, "Real-time evaluation of nMPRA CPU Architecture based on Multithreaded Execution," in *8th International Conference on Computer Science and Information Technology*, Amsterdam, Netherlands, 10–11 Dec. 2015.
[16] N. C. Gaitan, I. Zagan, and 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.