Implementation of Multi-Core Processor Based on PLASMA (most MIPS I) IP Core

Bojan Gruevski, Aristotel Tentov, Marija Kalendar
SS. Cyril and Methodius University - Faculty of Electrical Engineering and Information Technologies
Karpos II bb, PO Box 574, 1000 Skopje, Macedonia
E-mail: gruevski.bojan@gmail.com, {toto, marijaka}@feit.ukim.edu.mk

Abstract—**Multi-core processors is a design philosophy that has become mainstream in scientific and engineering applications. Increasing performance and gate capacity of recent FPGA devices has permitted complex logic systems to be implemented on a single programmable device. By using VHDL here we present an implementation of one multi-core processor by using the PLASMA IP core based on the (most) MIPS I ISA and give an overview of the processor architecture and share the execution results.**

Keywords: **FPGA, multi-core processor, MIPS, PLASMA, core, implementation, architecture, design.**

I. Introduction

The concept of parallelism plays important role in the computer industry and the scientific community because it defines a way for the computer to realize two or more operations simultaneously and as a result the applications or calculations that are performed usually will finish and give the desired results more quickly. The improvement of the parallelism is one of the most important methods for faster task executions and usually the most applicable results of these parallelism techniques can be found in the processors. For example of a parallelism let's say we transfer two equally large files on a disk [1]. The transfer speed will depend on the bandwidth of the disk. If they simultaneously were transferred on two disks separately then we would have double transfer bandwidth so the transfer will complete twice as fast as in the first case.

There are different types of parallelism and different forms to execute it. One example of usage of the parallelism concept on system level is the usage of multiple disks and processors. The workload to meet customer requests can be distributed between the processors and disks so the result will be a better responsiveness of the system. When a system can expand its memory and its number of processors we call this system a scalable system and this is a highly valued property of the server systems.

Other type of parallelism concept is the concept of instruction level parallelism (ILP) [2]. This concept refers the possibility of execution of more than one instruction in one program at the same time. These instruction may be consecutive or not. One of the simpler forms for achieving this is the use of pipelining which is a technique that was used in the first processors for achieving the effect of parallelism. The basic idea behind this technique is to overlap the execution of the instructions so it can reduce the execution time needed for a sequence of instructions. A key factor that leads to pipelining is the fact that not all instructions depend on the immediately preceding instruction so therefore they are partially or completely executed in parallel as much as possible.

II. Multi-Core Processors

Regardless of whether the processor core takes advantage of the instruction or thread level parallelism, multi-core processor have the advantage to exploit parallelism at task level found in the applications. A key factor that gives this advantage lies in the architecture of the task itself. Namely, each task has its own program counter, its own address space, its own registers and its own context so different tasks can be executed on different cores in the multi-core processor at virtually the same time [3]. This fact gives tremendous performance boosts on the execution of the applications but it is only achievable by good programming principles and very smart compilers that can order the code so it can be executed on a multi-core processor. If these requirements are not met then the performance of the execution of the application can be same or worse from the performance of the execution of the same application on a single-core processor.

III. PLASMA (Most MIPS I) IP Core

The PLASMA microprocessor is a 32 bit RISC processor core designed in VHDL[1] that is fully synthesizable. It executes all the instructions in user mode specified in the MIPS ISA with the exception of the instructions for misaligned memory access[2] [5]. On [4], the source code of the latest version of the PLASMA core can be downloaded. The current implementation incorporates and some peripherals like UART and Ethernet. It is tested on several FPGAs from [6] and [7].

[1] VHDL presents an acronym that consists from two acronyms: VHSIC and HDL. VHSIC stands for Very High Speed Integrated Circuit and HDL stands for ardware Description Language. VHDl is a language defined by the IEEE (ANSI/IEEE 1076-1993).

[2] The implementation of these instructions was not possible due to the fact that those instructions were patented during the design of the processor.

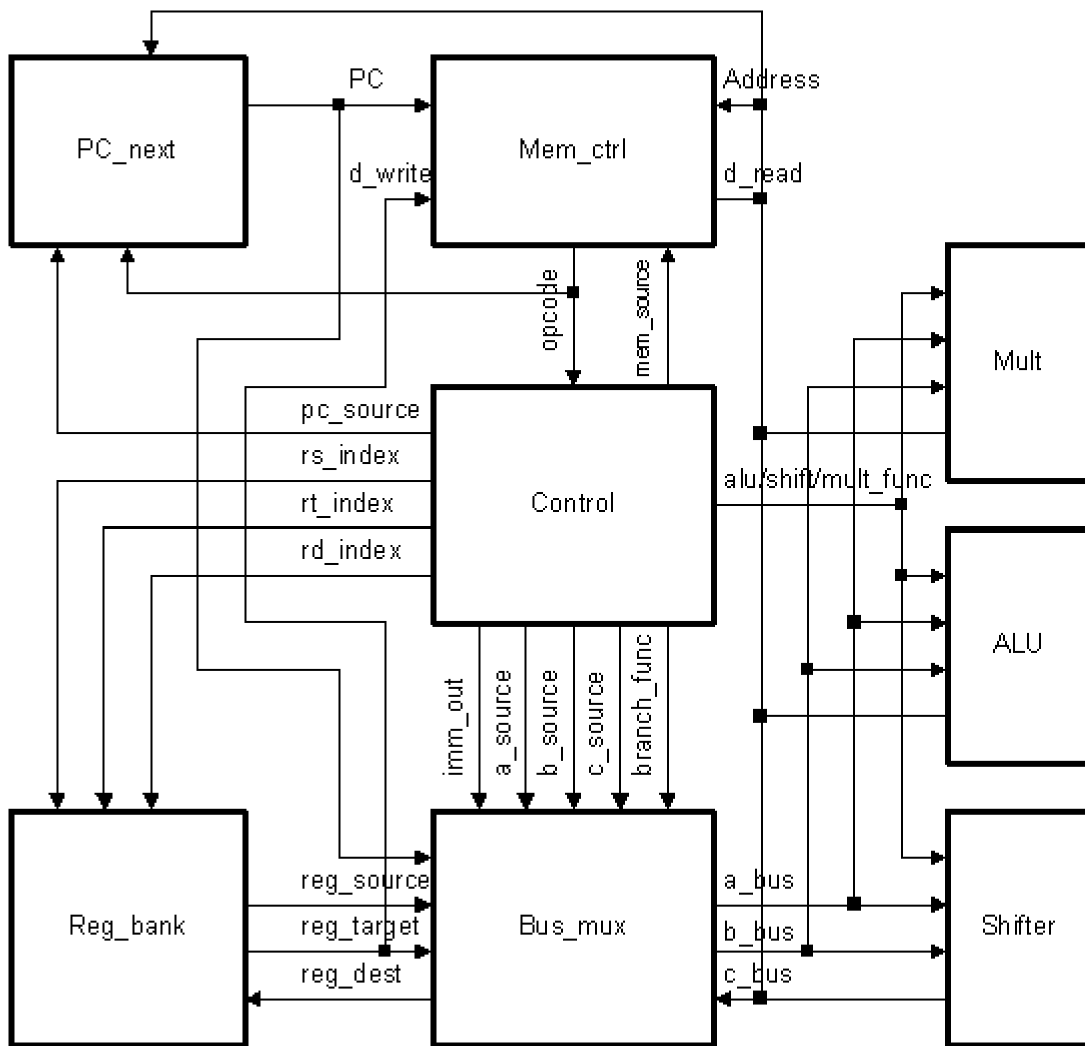


Fig. 1. PLASMA core architecture [1].

The processing unit can be implemented two or three stage pipeline and optional 4kB cache memory. The working frequency when implemented in [6] or [7] FPGAs has reached 25 MHz and 50 MHz depending of the FPGA model. The core is not implemented on a silicon level or at least it was not reported in the consulted bibliography.

Fig. 1 shows the architecture of the PLASMA core. For example an ADD instruction will perform the following steps (stages): the “pc-next” entity will pass the value of the program counter (PC) to the “mem_ctrl” entity which in return will fetch the opcode from memory (end of stage 1); the memory wil return the opcode (end of stage 2); the entity “mem_ctrl” will pass the opcode to the “control entity”; this entity will convert the 32 bit opcode to a 60 bit VLIW[3] code and will send control signals to the other entities; based on the “rs_index” and “rt_index” control signals, the entity “reg_bank” will send the 32 bit “reg_source” and “reg_target” values to the entity “bus_mux” (end of stage 3); based on the “a_source” and “b_source” control signals, the entity "bus_mux" multiplexes “reg_source” onto “a_bus” and “reg_target” onto “b_bus”; based on the “alu_func” control signals, "alu" adds the values from “a_bus” and “b_bus” and places the result on “c_bus”; based on the “c_source” control signals, "bus_mux" multiplexes “c_bus” onto “reg_dest”; based on the “rd_index” control signal, "reg_bank" saves “reg_dest” into the correct register (end of stage 4, stage 3 if using two stage pipeline); read or write memory if needed (end of stage 5, stage 4 if using two stage pipeline).

[3] VLIW is an acronym that stands for Very Large Instruction Word

The result of the simulation of the operation of the PLASMA core is shown on Fig. 2.

IV. Multi-Core Processor Based on Plasma IP Core

The multi-core processor showed on Fig. 9 and 10 that was implemented is a four (or two) core processor that is based on the PLASMA IP core mentioned before. It consists of:

Four cores based on the PLASMA IP core: There was very little core modification so that the architecture of the multi-processor remains simple. The only modification was with the addition of an arbiter sub module that will generate requests to the main bus arbiter in order for the processor to start read/write memory cycle. Bus arbiter (Fig 3): this is the module that controls the bus. It assigns the bus to one core at a time so it eliminates bus conflicts between the cores [8]. The main job of this entity is to make sure that the cores will access the bus fairly. For example core zero has bigger priority than one, one has bigger priority than three, three than four and so forth. When the bus will be freed by a core, the arbiter assigns the bus to the next core from the FIFO list that has requested it. Fig. 4 shows the usage of the arbiter.

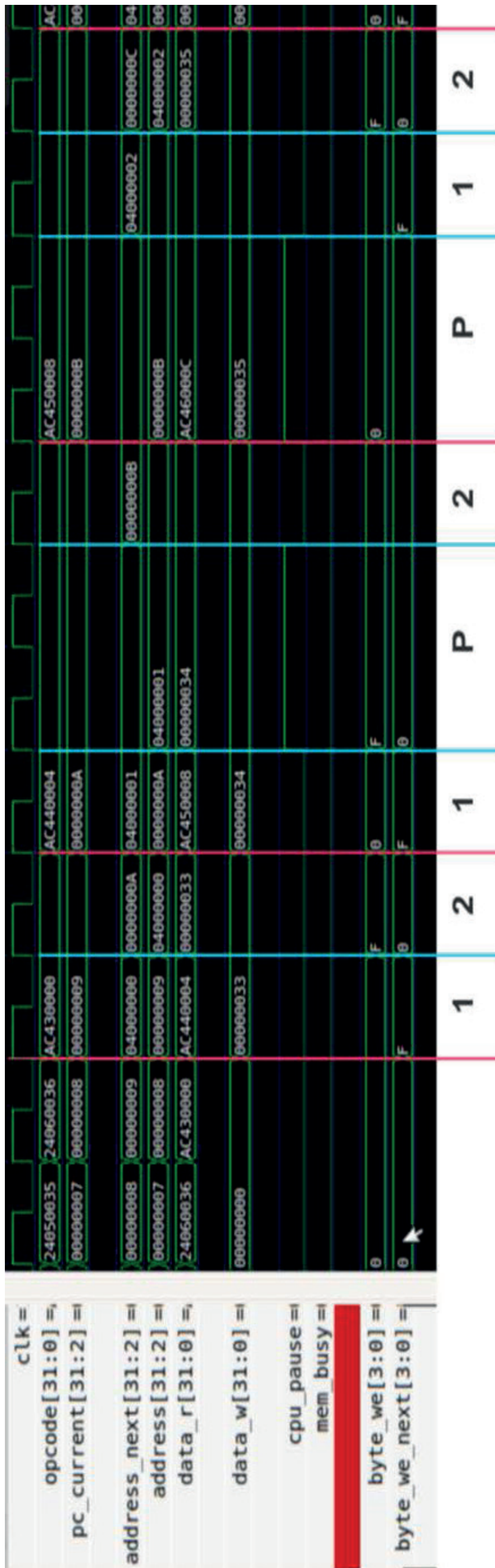


Fig. 2. Operation of the PLASMA core.

If the bus is free, this module will provide access on it on the first core that will make request. Subsequent request (while the bus is in use) are put in a FIFO list in the order they happen. If there are simultaneous requests the priority is given by the core number.

The arbiter's buffer has to be with size equal or larger than the number of cores to accommodate the worst case scenario - when all cores have requests for bus utilization. His work is showed on Fig. 4.

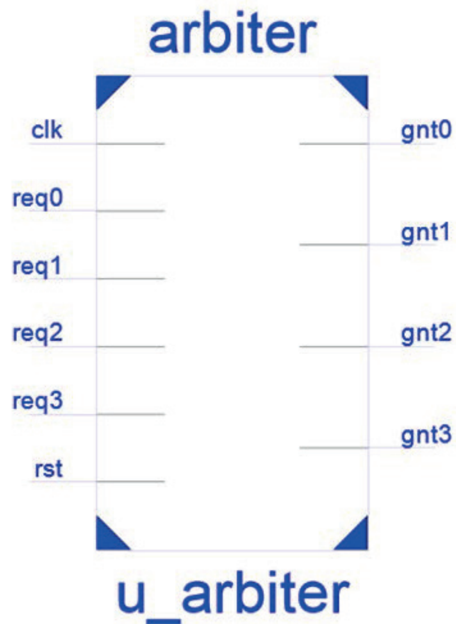


Fig. 3. Architecture of the arbiter.

Bus multiplexer: depending on which core the bus is allocated, the multiplexer connects the selected core bus with the main bus [9]. Its architecture is shown on Fig. 5.

There are several peripherals on the processor:

1. UART - standard UART controller which can be accessed by two registers, one for received data, and the other for data transmission. The data is consisted from 8 bits and it is without any parity bit.

2. Registers for interrupt requests - two registers, one for reading the status of the pending interrupts and the other for interrupt masking.

3. Counter register - 32 bit register used for counting clock cycles. It increases its value on every cycle

4. GPIO ports - two registers, one used to write values to the GPIO pins and the other used to read values from the GPIO pins.

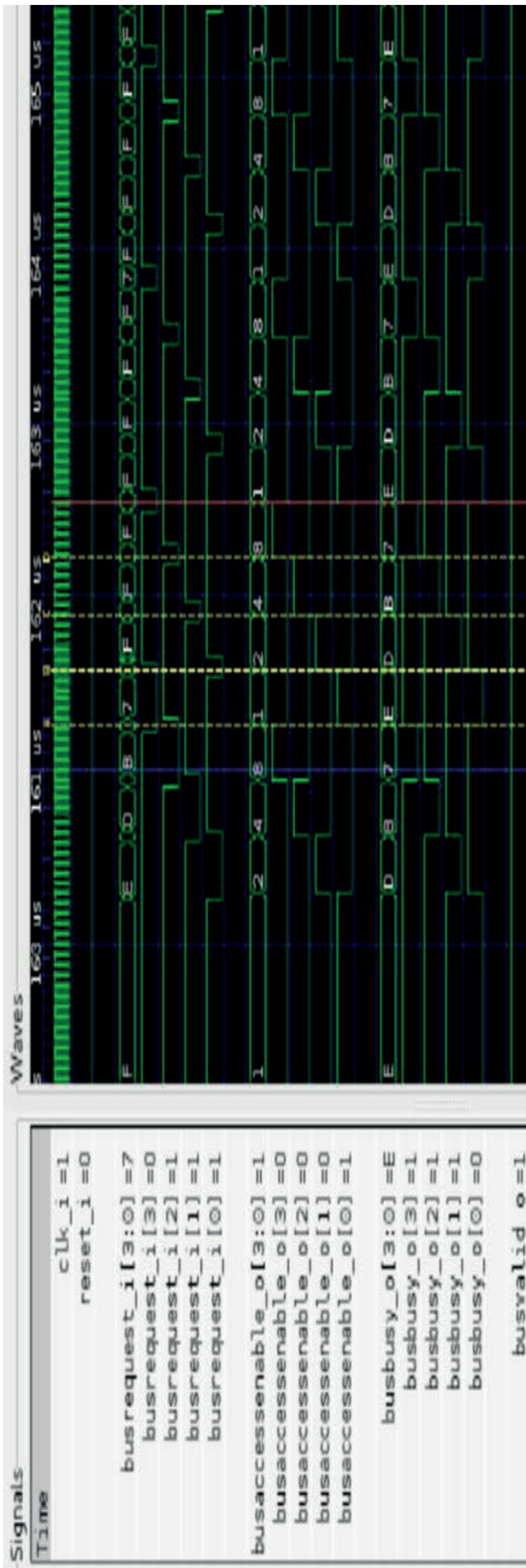


Fig. 4. Operation of the arbiter.

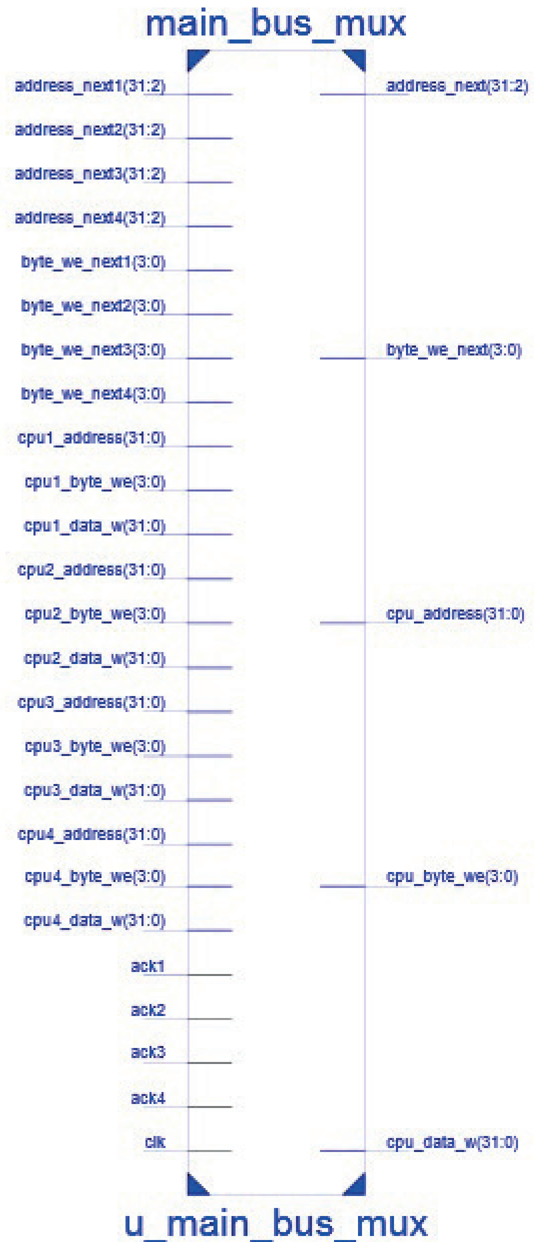


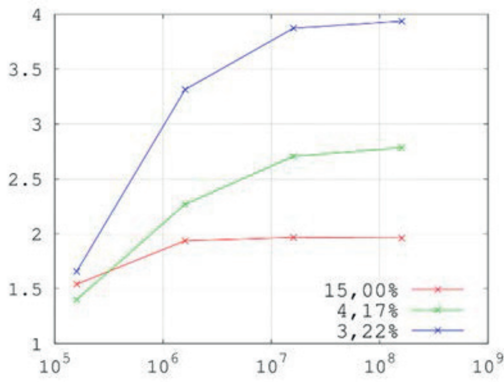
Fig. 5. Bus multiplexer architecture.

V. Results

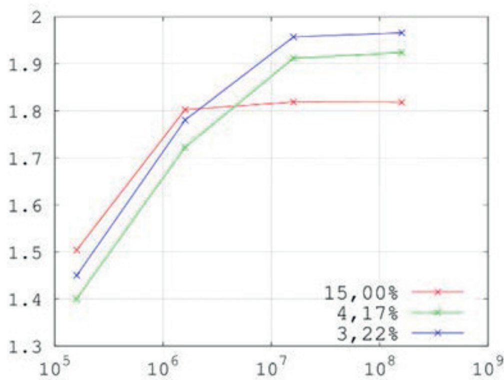
The following findings are based on test runs of different programs and in every succeeding program the number of instructions was increased [10]. Fig. 6 shows how the performance improvement in processing performs with increasing amount of work for the tasks and it is measured by the number of instructions that the tasks are running. The result shown are for three different tasks where the percentage of instructions accessing memory changes. In the plot on Fig. 6 tasks that have 15%, 4.17% and 3.22 % access to the processor bus are shown. The results from Fig. 6a and 6b are comparison of a processor with four and two cores respectively. In all cases it is seen that the performance rises with increasing the work that the task are doing. This is because the percentage of time it takes the operating system to reschedule tasks is declining. Beyond a certain value this time becomes negligible compared to the time of actual processing. It is also noticeable that we always get a limit on the improvement which depends on the percentage of instructions that access the bus. In the best

case on Fig. 6a this value is about 4 whereas on Fig. 6b is 2 which are the respective values that theoretically we can achieve in ideal cases. The ideal case is when we have no collisions on the bus and the only time we can assure that this will be possible is when no core will access the bus at any moment which is impossible. The increase in collisions causes degraded performance.

The next findings on Fig. 7 are based on testing the processor, running programs in which the utilization is increased progressively on the bus [11]. Collisions on the bus increase with increasing the number of cores and with increasing the bus utilization every time we add a core. Evaluation is done on how the efficiency of the core increases as the collisions decrease. On Fig. 7 it is shown that the improvement in the processing time of the tasks decreases along with the increase in the percentage of instructions that access the bus (for unwanted results, which is very low percentage, the improvement decreases again). This is because for these small values, effective processing time of tasks begins to decrease and the time to reschedule the tasks is no longer negligible.



(a) Execution on four cores



(b) Execution on two cores

Fig. 6. Improvement in execution time when going from one to four cores. Shows the relationship of times of execution in function of the amount of processing tasks are run. The amount of processing measuring instructions that execute tasks.

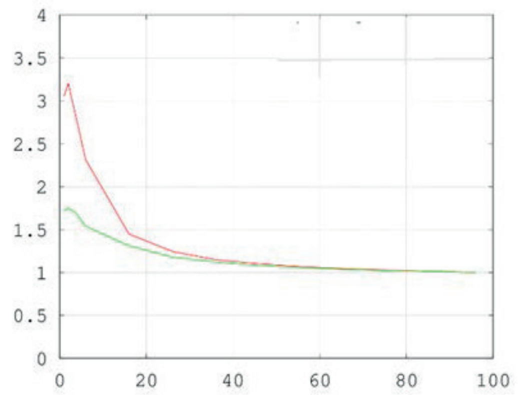


Fig. 7. Improvement in execution time for four and two cores, compared to a core function of the percentage of instructions that access to the bus (red - 4 cores, green - 2 cores).

VI. Conclusion

In this paper we have presented an implementation of a multi-core system on which were performed several tests. The presented result shows dependency of performance with the number of cores, the type of application (task) and whether we have memory cache or not. The factors that determinate the main efficiency of a processor is the bus usage on behalf the processing that needs to be done.

There are other factors to be evaluated in the future including other type of arbiter policies and memory cache that is not shared between data and instructions. Other future work may involve implementation of a larger multi-core processor with more than four cores, development of an operating system that will control the proper distribution of the tasks that need to be executed. Here we have showed author’s first implementation of a multi-core processor and there are a lot of possibilities for further development of the design.

VII. APPENDIX

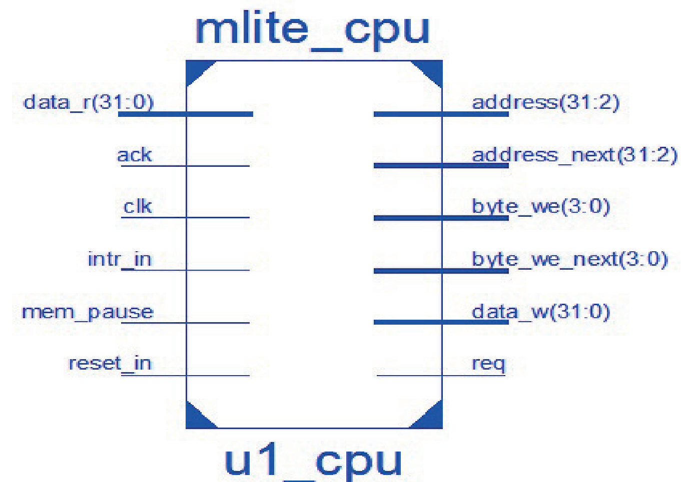


Fig. 8. PLASMA core connections.

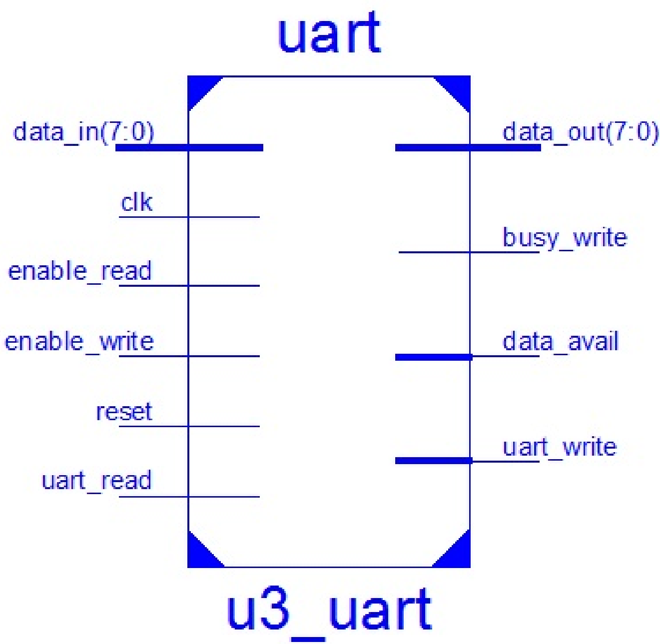


Fig. 9. Multi-core processor UART interface.

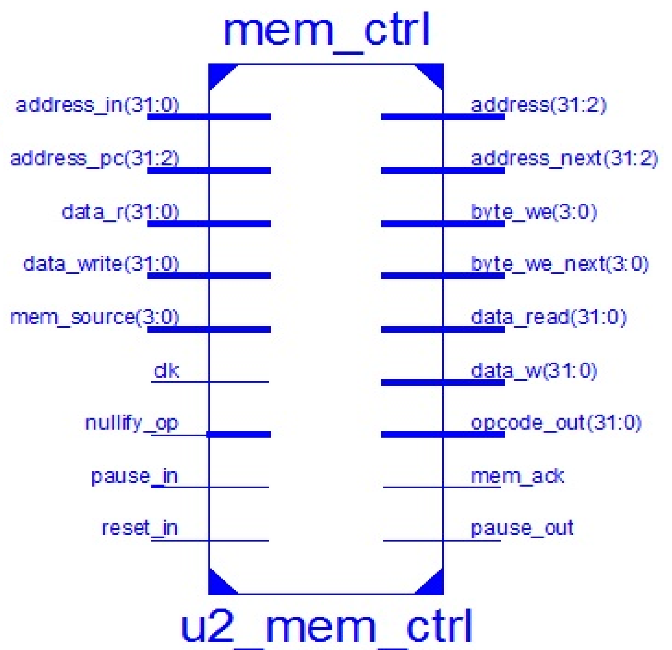


Fig. 10. Memory controller of the multi-core processor.

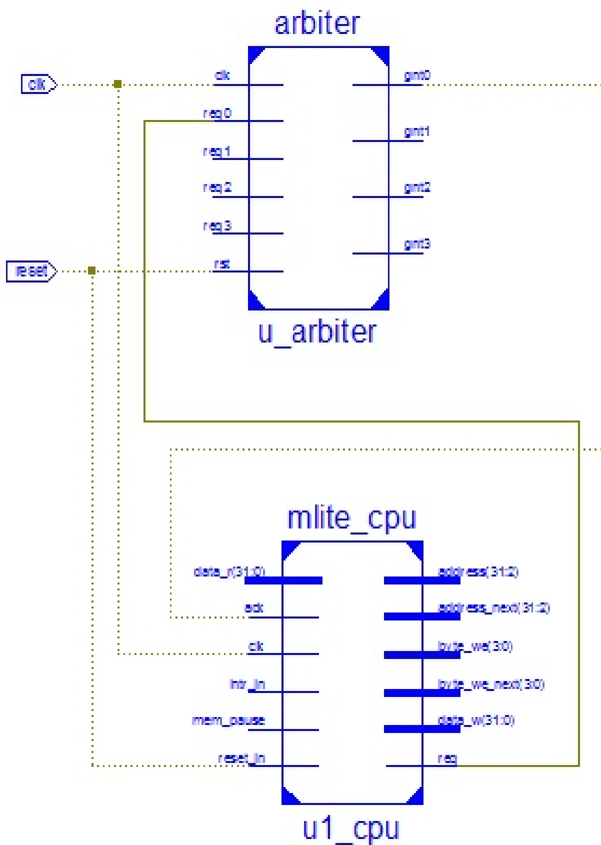


Fig. 11. Connection of the bus arbiter with one of the cores.

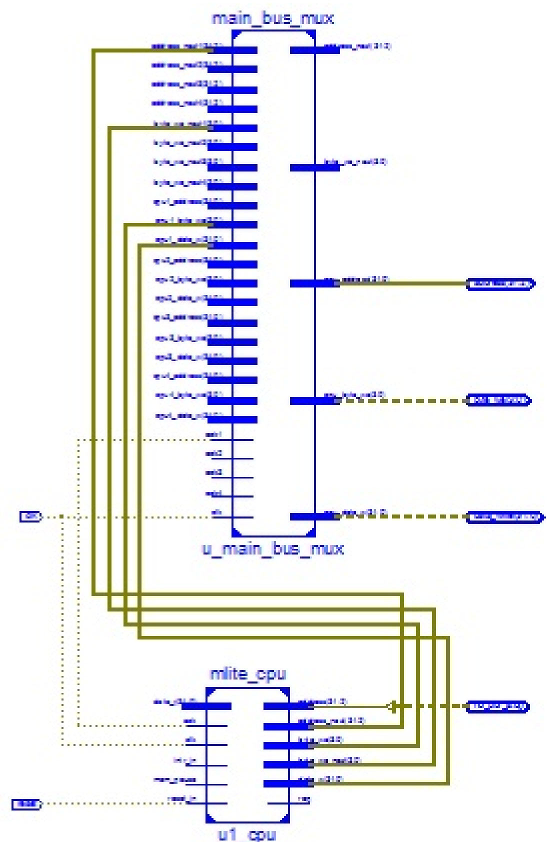


Fig. 12. Connection of the bus multiplexer with one of the cores.

REFERENCES

[1] M. J. Flynn, “Computer Architecture: Pipelined and Parallel Processor Design,”. Jones & Bartlett Learning, Sudbury, Massachusetts, 1995.

[2] J. L. Hennessy, D. A. Patterson, “Computer Architecture - A Quantitative Approach,” 4th Edition. Morgan Kaufman Publishers, Burlington, Massachusetts, 2007.

[3] A. S. Tanenbaum, “Modern Operating Systems,” 2nd edition. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[4] Plasma – most MIPS I (TM) opcodes: Overview [Online]. Available: http://opencores.org/project,plasma,overview, (October, 2013)

[5] MIPS M51xx Warrior-M class CPU Core [Online]. Available: http://www.mips.com/, (November, 2013)

[6] Xilinx [Online]. Available: http://www.xilinx.com/, (October, 2013)

[7] Altera [Online]. Available: http://www.altera.com/, (October, 2013)

[8] David A. Patterson, John L. Hennessy, “Computer Organization and Design - The Hardware/Software Interface”, 3rd Edition. Morgan Kaufman Publishers, Burlington, Massachusetts, 2005.

[9] J. M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic “Digital Integrated Circuits - A Design Perspective”, 2nd Edition, Prentice-Hall, Upper Saddle River, New Jersey, 2002.

[10] D. Silva, K. Stangherlin, L. Bolzani, F. Vargas, “A Hardware-Based Approach to Improve the Reliability of RTOS-Based Embedded Systems,” IEEE Computer Society, 2011, p. 209.

[11] J. Tarrillo, L. Bolzani, and F. Vargas, “A Hardware-Scheduler for Fault Detection in RTOS-Based Embedded Systems,” Digital System Design, Architectures, Methods and Tools, 2009, pp. 341 – 347.