Replacing the SPARC-based core of the Leon3 HDL microprocessor model with a MIPS-based core.

Arvanitakis Ioannis
jarvanitakis@gmail.com

Mamalis Dimitrios
dimitris.mamalis@gmail.com

Abstract

The aim of this study is the creation of a MIPS architecture microprocessor with full potential of operating and controlling its functions, and its implementation in a Field-Programmable Gate Array (FPGA). In order for this to become possible, we relied on the already implemented SPARC architecture microprocessor, Leon3 by Gaisler Research, which provides adequate functional units that add potentials missing from the implemented open source MIPS models.

At first, the detailed operation of the two microprocessors is presented and we conclude in composing a flexible microprocessor, in which the good functionality of Leon3 (caches, AMBA bus etc.) is combined with the advantages of the MIPS instruction set architecture. The final circuit is granted in full compatibility with the FPGA implementation circuit models. This microprocessor can be used in various applications and provides a lot of potential for further development for educational and research purposes.

1. Introduction

The reason for this study was the lack observed in the existence of MIPS architecture models of open source and good functionality. Most models which are freely provided have inadequate potential of operating their functions.

We worked on a complex processor model, like Leon3 and we successfully replaced the SPARC architecture core with an MIPS architecture core, based on the MIPS789 model which is freely available in verilog-2001 hardware description language.

The complexity of this study lies in the difficulty of adapting a fully functional processor to a core architecture which, although at first seems to be of similar philosophy (both of RISC type), proves in fact to handle very differently its interconnection to the elements outside the core, which are necessary for its operation.

Such elements are for example the caches, the register file and the AMBA bus through which the main memory (RAM) and the peripheral devices controllers are interconnected.

The final outcome of the study is a functional MIPS processor programmed in FPGA.

2. Instruction set architecture

The instruction set architecture is a list of all the commands that a processor can execute. There are two basic types of such architecture: CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing). In CISC we encounter a large number of complex instructions which are executed in more than one machine cycles. On the contrary, in RISC the instructions are fewer, simple and executed in one machine cycle.

Table 1: Basic features of CISC and RISC architectures.

CISC	RISC
Emphasis on hardware	Emphasis on software
Multi-clock instructions	Single-clock instructions
Embedded LOAD and STORE	Independent LOAD and STORE
Small code size, low cycles per second	Large code size, high cycles per second
Transistors for decoding complex instructions	More transistors on registers

2.1. RISC Architecture: SPARC vs. MIPS

In this paragraph, we will make a brief reference to the two different models of RISC architecture, which we are going to deal with later in this study.

Starting with the SPARC architecture (Scalable Processor Architecture), we should mention the basic structures of a SPARC processor. This includes an integer unit (IU), a floating-point unit (FPU) and, optionally, a co-processor (CP). Each unit has its own registers. The SPARC instructions can be categorized into six groups: load/store, integer arithmetic, control transfer, register/record control, FPU functions, CP functions. Special procedure call instructions use the windows organization of the IU register file.

Respectively, in MIPS architecture, the processor includes the core, an FPU and optionally a CP. Each unit has also its own registers. The MIPS instructions are categorized based on their pattern into three major groups: the R-type instructions with solely direct register operands, the I-type instructions with an immediate operand or offset and the J-type which are jump instructions with direct address operand.

In both architectures no instruction can be more than 32-bit wide, since every instruction must fit in the pipeline and be completed in a single-clock cycle.

3. Leon3 Microprocessor

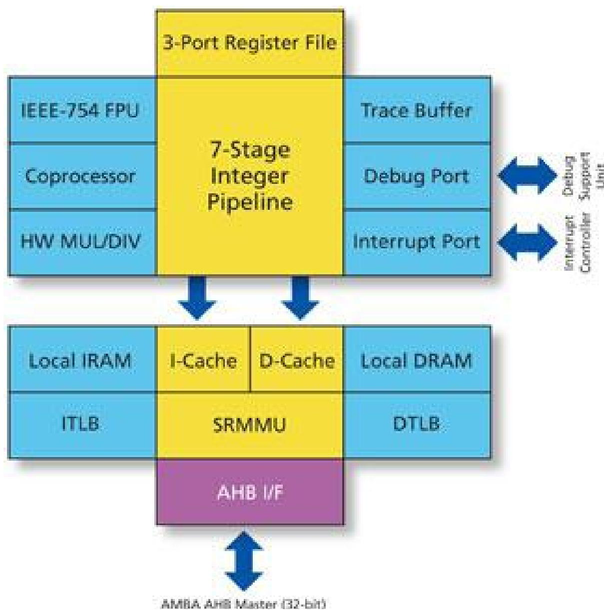
In this paragraph, we are going to present in brief the structure and the characteristics of Leon3 microprocessor, with which we are going to deal.

Leon3 is an open-source implementation of a 32-bit microprocessor in VHDL hardware description language. Leon3 model is designed for embedded applications. Its basic features are the seven pipeline stages, and the separate units of multiplication and division, floating point, memory management unit and MAC function unit. Also, it includes implemented interfaces for AMBA 2.0 AHB bus, coprocessor and on-chip debugging. The integer unit, the general purpose register file, the caches and their controllers, along with the floating point unit and the coprocessor are all regarded as the processor's core.

The integer unit is what interests us the most, since it is here that both the Leon3 pipeline structure and the cache communication are implemented.

The pipeline is implemented in the following seven stages: Instruction Fetch, Decode, Register Access, Execute, Memory, Exception, Write.

The cache subsystem is implemented on the basis of Harvard architecture, in two separate cache units of instruction and data.



Picture 1. Leon3 core diagram.

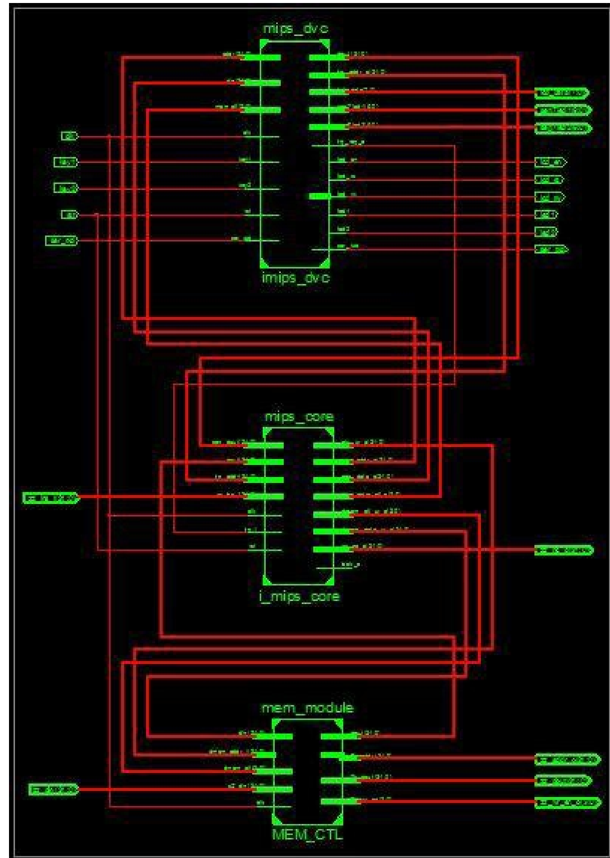
4. The MIPS789 microprocessor

The MIPS789 microprocessor is freely available through the OpenCores community. It is a core written in verilog-2001 hardware description language. It supports almost every MIPSI instruction with a pipeline structure of five stages.

It consists of the mem_array and mips_sys units. The mem_array is a unified external memory of instructions and data and the mips_sys contains the core and the coprocessor. An elementary register file, the interrupt controller and a data memory, which operates as a vestigial data cache, are all implemented in the mips_sys unit.

From the aforementioned units, we are solely using the core, and for the rest we rely on the functionality of Leon3.

It is worth noting that the MIPS789 pipeline mechanism differs from the typical pipeline structure which was developed by John L. Hennesy's team and is used in the MIPS models. More specifically, the MIPS789 has implemented a unified instruction fetch and decode stage. The register read stage follows the previous one and it is not done along with the instruction decode. Finally, the branch control is incorporated in the register read stage and not the instruction execution stage.



Picture 2. MIPS789 core.

5. Conversions and connection of the two microprocessors.

The targets and modifications on the MIPS789 structure will be briefly presented below. The basic target is to use only the absolutely necessary units of MIPS architecture. As a result, the systems of the vestigial cache, the coprocessor as well as the register file were removed. The MIPS architecture core which came up, uses the cache unit of Leon3 itself, as well as its register file, without making any alterations in their structure. In order for something like that to become feasible, it was necessary to study the internal structures of the MIPS789 and Leon3, the signals they generate, the timing of the overlapping stages and plenty of other problems which arise during the modification of all the above. Finally, the procedure of connecting the two, already modified, systems will be described.

5.1 Units of data and instructions cache

As it was briefly outlined above, the MIPS789 contains a unit of vestigial data cache. This unit was removed and the suitable connections were created, so that the final core solely uses the instruction and data cache of the Harvard architecture that Leon3 has. In tables 2 and 3 some of the most important signals which were modified or cut off are presented.

Table 2. I-cache signals

	LEON	MIPS	Short Description
I-cache Input	inull	0	Instruction Nullify
	rpc	pc_next	Raw pc
	fpc	zz_pc_o	Fetch pc
	fbranch	branch	Instruction branch
	rbranch	branch	Instruction branch
	fline	ifline = 29'b0	Flush line offset
	flush	0	Flush icache
	flushl	0	Flush line
I-cache Output	data	zz_ins_i	Data type
	set	iset	Set data type
	hold	hold	Hold when ready
	flush	--	Flush in progress
	mds	imds	Memory data strobe
	idle	--	Idle mode

Table 3. D-cache signals

	LEON	MIPS	Short Description
D-cache Input	asi	asi_code	Address space identifier
	maddress	alu_ur	Memory address
	eaddress	eaddr = 32'b0	Execute Address
	edata	dmem_data_ur	Execute Data
	size	size	Data Size
	enaddr	dmem_ctl_ur(2)	Enable Address
	nullify	0	Nullify
	lock	dmem_ctl_ur(4)	Lock cache
	read	dmem_ctl_ur(1)	Read cache
	write	dmem_ctl_ur(3)	Write cache
	flush	0	Flush Cache
	flushl	0	Flush line
	dsuen	dmem_ctl_ur(0)	DSU enable
D-cache Output	data	dout	Data type
	set	dset	Set data type
	hold	hold	Hold cache
	mds	dmds	Memory data strobe
	idle	--	Idle mode

5.2. Coprocessor unit

Following the same rationale, the MIPS789 coprocessor unit was removed and its signals were cut off, since the final core should be able to use the interconnection for coprocessor units that Leon3 provides.

5.3. Register file

Subsequently, the MIPS789 register file was removed, in order to be replaced by the Leon3 register file. A basic adaptation made was the modification of the input signals from 8-bit to 5-bit. This occurred because the MIPS does not use register windows, but a fixed file of 32 registers.

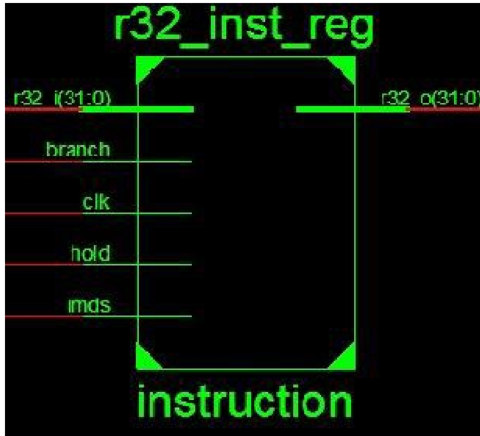
The following table presents the correspondence of the Leon3 register file signals with those of the MIPS core.

Table 4. Register file signals.

	LEON	MIPS	Σύνοημη Πεπιγπαθή
Input	Wdata	Data	Write Data
	Waddr	Wraddress	Write Address
	we	wren	Write Enable
	raddr1	rd_address_a	Read Address
	re1	1	Read Enable
	raddr2	rd_address_b	Read Address
	re2	1	Read Enable
	rclk	clk	Read Clock
	wclk	clk	Write Clock
Output	rdata1	qa	Read Data
	rdata2	qb	Read Data

5.4. Instruction register

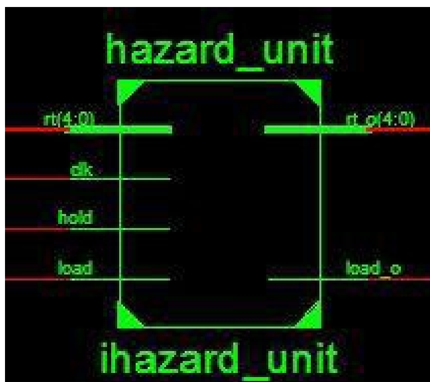
The lack of a register which could maintain the instruction coming to the core was creating problems to the timing of the core and the caches. Therefore, such a register was added, controlled by the same control signals as the caches. The remaining registers are also using these signals, in order to achieve the best timing possible.



Picture 3. Instruction register.

5.5. Data hazard control

During the study of the MIPS789 pipeline, we noticed a wrong implementation of the processor stall mechanism in the case of data hazards. More specifically, the read-after-write dependence in the event that two consecutive instructions are using the same register, when the first instruction is a memory load, was not being recognized. Thus, we were led to the addition of extra control to detect and correctly handle such cases.



Picture 4. Data hazard control unit.

5.6. Connection of the two projects

As it has already been mentioned, the two processors are written in different hardware description languages. In order to connect them in a joint project file, it was necessary to have a VHDL top-file, where the mapping of all the in-between signals is done, and to create the suitable VHDL components.

6. Simulation, debugging and composition of the final project

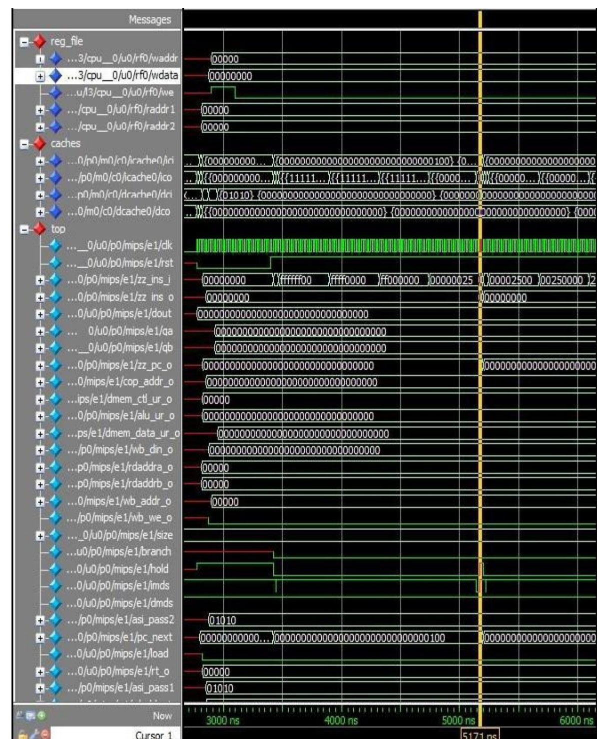
For the final simulation and debugging of the project, we relied, as much as possible, on the simulation techniques available for the Leon3 model. After studying its testbenches, we concluded that they initialized the processor and booted the programs to be executed through two srec files. These are ASCII data files of a binary coding system. The first file, prom.srec, contains the processor's initialization code. This code was modified properly, to be compatible with the MIPSI instruction set. The desired execution code was placed in a similar coding in the second file, sdram.srec.

Subsequently, these files together with the processor's final layout, are loaded and executed with the Modelsim 6.3f program. There, one can observe the amount of signals produced in the timeline. By giving suitable parts of critical code, the correct operation of the microprocessor we created is confirmed. In the following paragraphs, we are going to give in brief some examples of executing critical instructions. This is about a general snapshot of an execution, an execution of a logical instruction, of a loading instruction and finally, of a branch instruction.

6.1. General snapshot of execution

In picture 5, some of the most important signals during the processor's boot and the execution of the first two instructions of prom.srec code can be noticed. The initial signal values and their alternations are shown in this particular snapshot. More specifically, the clk has a period of 25ns (that is frequency of 40MHz) and we can see the correct signal change concerning the pc (pc_next, zz_pc_o), the consecutive changes of the signals concerning each instruction word (zz_ins_i, zz_ins_o) and the register file access signals. Most of these signals will be analysed in detail in the following paragraphs, where their functionality per instruction is shown.

It is worth referring to the case of signal hold (of negative logic), which takes value from the caches in order to stall the processor in case of miss. Here, it is clearly shown that the instructions pass through and the pc changes are being done during the time when the hold has the value "1". When its value is "0", nothing happens in the processor's core!



Picture 5. General snapshot of execution.

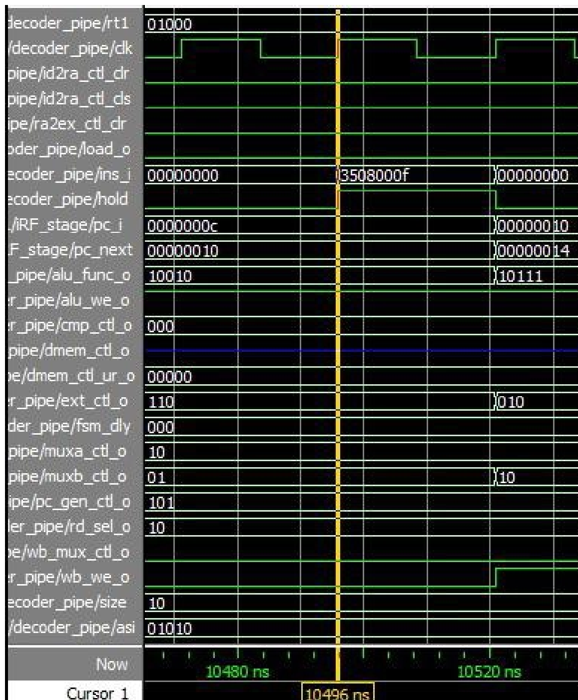
6.2. Snapshot of executing a logical instruction

In this paragraph we are going to follow the process of executing an instruction

ori $t0, $t0, 0xf

which is coded in the hexadecimal 0x3508000f.

The instruction reaches the rising of signal hold through the ins_i signal in the secode stage. During the fall of hold the relevant signals are produced (picture 6). More specifically, as demonstrated in picture 8, the instruction following the register file access has successfully written into register 8 ($t0) the value 0x0081000F after three pipeline stages and with the use of forwarding, given the fact that the previous instruction produces the value 0x810000 for the same register. The forwarding mechanism is portrayed in picture 7, where, through the fw_alu and the ext_i, which is the immediate operand, the result of the alu (alu_ur_o) is the correct result and this is what is written into register 8.



Picture 6. Arrival of the instruction ori $t0, $t0, 0xf at the decode stage and its execution.

Picture 7. Forwarding the value of register 8 for the instruction ori $8, $8, 0xf.

Picture 8. Accessing the register file for writing by the instruction ori $t0, $t0, 0xf.

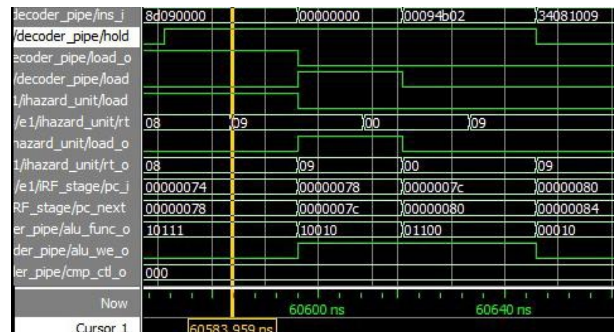
6.3. Snapshot of executing a load instruction

The next case is that of a load instruction

lw $t1, 0($t0)

which reaches the core encoded in the hexadecimal 0x8d090000.

In this case, the concern is on different signals, such as the load_o signal, which indicates that a load instruction preceded and which passes through the hazard control unit from the decode stage, as well as the rt_o signal, which carries the address of the register where the loading will take place. In picture 9 we see the load_o signal having already taken the value ‘1’ from the decode stage and through the hazard unit is transferred to the next pipeline stage, as well as the value of the rt register (here $t1, which is 9). After the arrival of the next instruction (0x00000000), the load_o signal is back to ‘0’, and since the ensuing instruction is a nop, the control for read-after-write hazards does not stall the processor.



Picture 9. Arrival of the instruction lw $t1, 0($t0) at the decode stage and at the hazard unit.

Following, in picture 10, we observe the activation of enaddr, eenaddr and read signals, which allow the read of the memory address, which was earlier calculated by adding the offset with the content of the rs register. The value written into register 9 is ‘0’, as shown.

Picture 10. Value read from the data cache.

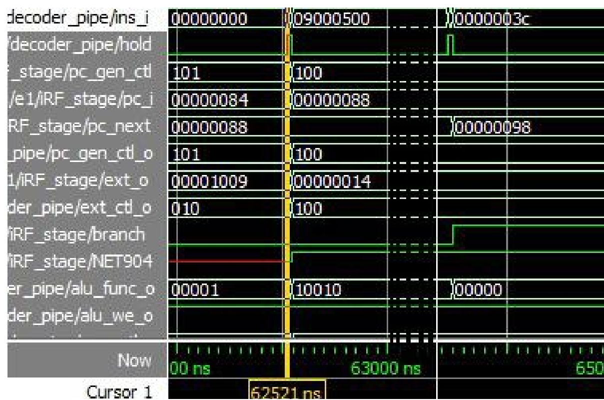
6.4. Snapshot of executing a branch instruction.

Below, we are going to go over the execution of a branch instruction and in particular that of

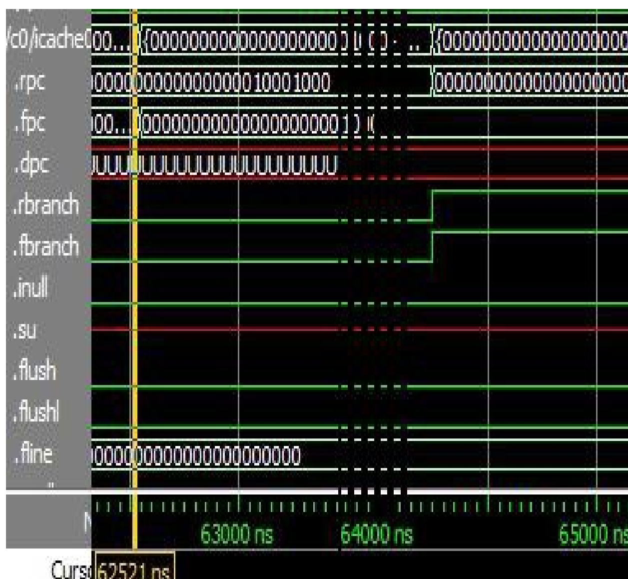
bne $t0, $t1, <L>

which reaches encoded in the hexadecimal 0x15090005.

The instruction arrives through the zz_ins_i, it is decoded and then passes through the control of the conditional jump. The result of this process is transferred through the interior signal NET904, which takes the value ‘1’, since the condition is true. At the same time, the offset that must be added to the pc is transferred through the ext_o signal, so that the address where the control will be transferred can be calculated. As shown in picture 10, in the register file read stage, the branch signal has been activated, which in turn activates the fbranch/rbranch signals of the instruction cache, so that the proper instruction will be fetched next, as shown in picture 11. Moreover, we notice the conversion of the pc_next signal. Instead of increasing by 0x4, gets to the memory address where the <L> branch tag is found on the loaded code.



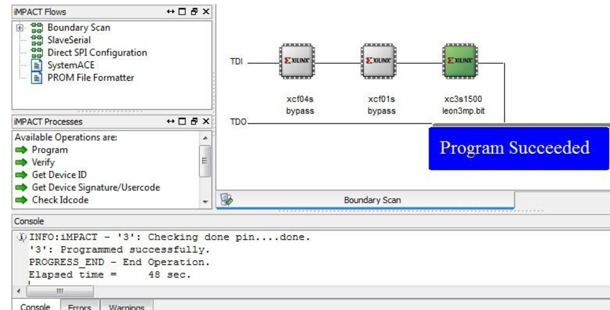
Picture 11. Execution of a bne $t0, $t1, <L> instruction.



Picture 12. Activation of the fbranch and rbranch signals in the instruction cache.

6.5. Composition process.

The composition of the final project and its programming in FPGA was accomplished through the use of Xilinx tools, found in the Xilinx WebPack. After the successful completion of the composition process, the final bit file is produced, with which we completed the FPGA programming on the gr-xc-3s1500 board by Pender Electronics, which was at our disposal.



Picture 13. Successful FPGA programming with the final design on the IMPACT tool.

```
Device utilization summary:
---------------------------
Selected Device : 3s1500fg456-4
Number of Slices: 5139 out of 13312 38%
Number of Slice F/Fs: 3177 out of 26624 11%
Number of 4 input LUTs: 9703 out of 26624 36%
Number used as logic: 9700
Number used as Shift registers: 3
Number of IOs: 264
Number of bonded IOBs: 138 out of 333 41%
IOB F/Fs: 158
Number of BRAMs: 17 out of 32 53%
Number of GCLKs: 2 out of 8 25%
Number of DCMs: 2 out of 4 50%
================================================
Timing Detail:
--------------
================================================
Default period analysis for 'clk':
Clock period: 28.538ns (frequency: 35.041MHz)
Total number of paths / destination ports:
3628222487 / 7957
------------------------------------------------
Delay: 35.673ns (Levels of Logic = 26)
Source: 3.cpu[0].u0/p0...s/q_4 (FF)
Destination: l3.cpu[0].u0/cmem...r0 (RAM)
Source Clock: clk rising 0.8X
Destination Clock: clk rising 0.8X
```

Picture 14. Extract from the composition process report.

7. Conclusions and future work.

On completing this study, we must refer to the conclusions we reached, as well as to the possibilities for further development of the particular project.

The final project is fully functional and compatible with the implementation techniques for FPGA circuits. However, the problems we faced, created a lot of ideas for future projects, to the implementation of which we are eager to provide our assistance and our experience.

More specifically, there is the possibility of interconnecting with other functional units of the Leon3 processor, such as the multiplier and division unit, the floating point unit and the coprocessor unit. Moreover, with the appropriate modifications, the interrupt control unit can be supported. One of the most useful parts of

Leon3 which could be connected is that of the debugging unit, in fact which would make the project manageable more easily, with the Leon3 control tools, which are provided freely by the manufacturing company, provided that they become compatible with the MIPS architecture. In order to do so, the operation of the AMBA bus should be studied, which could possibly be replaced by an open source bus, in which case a certain modification for the use in MIPS environment is necessary.

Finally, an important object of possible pursuit is the adaptation of one of the operating systems provided by Leon3 and the board xc-3s1500 in MIPS code. This will allow the use of peripheral units supported by the Leon3 environment, as well as the execution of complex benchmarks for the evaluation of the entire effort and the comparison with the SPARC architecture of the original system.

8. Bibliography.

1. David A. Patterson, John L. Hennessy: “Computer organization and design – The hardware/Software Interface – 3rd Edition”, Morgan Kaufmann, 2005
2. John L. Hennessy, David A. Patterson: “Computer Architecture – A quantative approach – 4th Edition”, Morgan Kaufmann, 2007
3. Sandi Habinc: “Lessons Learned from FPGA Developments”, Gaisler Research, 2002
4. Jiri Gaisler, Edvin Catovic, Marko Isomaki, Kristoffer Glembo, Sandi Habinc: “Grlib IP Core User‟s Manual”, Gaisler Research, 2008
5. Jiri Gaisler, Edvin Catovic, Sandi Habinc: “Grlib IP Library User‟s Manual”, Gaisler Research, 2008
6. Jiri Gaisler, Marko Isomaki: “Leon 3 GR-XC3S-1500 Template Design – Based on GRLIB”, Gaisler Research, 2006
7. “GR-XC-3S-1500 Deve3lopment Board – User Manual”, Gaisler Research/Pender Electronics, 2006
8. Jiri Gaisler Mailing List, sparc@yahoogroups.com
9. Lutz Buttelmann: “How to setup LEON3 VHDL simulation with Modelsim”, 2007
10. “The SPARC architecture manual – version 8” SPARC International Inc.
11. “Amba Specification, Rev, 2.0”, Arm Limited, 1999.
12. “ISE Release Notes and Installation Guide”, Xilinx Inc. 2004
13. “ISE In-Depth Tutorial” Xilinx Inc. 2004
14. “XST User Guide”, Xilinx Inc. 2004
15. “Fpga Editor Guide” Xilinx Inc. 2004
16. “Fpga Tutorial” http://www.fpga4fun.com
17. Miles J. Murdocca, Vincent P. Heuring: “Principles of Computer Architecture”, 1999
18. Sivarama P. Dandamudi: “Guide to RISC Processors for Programmers and Engineers”, Springer, 2005
19. “MIPS IV Instruction Set”, Revision 3.2, 1995
20. “Sparc 7 Instruction Set- Assembly Language Syntax”, Atmel Inc., 2002
21. Dominic Sweetman: “See MIPS run Linux – Second Edition”, Morgan Kaufmann, 2007
22. D. Nikolos: “Computer Architecture”, University of Thessaly Publications, 2000.