Ninth LACCEI Latin American and Caribbean Conference (LACCEI'2011), Engineering for a Smart Planet, Innovation, Information Technology and Computational Tools for Sustainable Development, August 3-5, 2011, Medellín, Colombia.

A CPU Simulator based on FPGA soft processor

Raul R. Peralta Meza
Universidad Catolica San Pablo, Arequipa, Peru, rperalta@ucsp.edu.pe

Cesar A. Vera Bernal
Universidad Catolica San Pablo, Arequipa, Peru, cesar.andres.vera@ucsp.edu.pe

Francisco J. Guerra Manchego
Universidad Catolica San Pablo, Arequipa, Peru, francisco.guerra@ucsp.edu.pe

Paola T. Llerena Valdivia
Universidad Catolica San Pablo, Arequipa, Peru, paola.llerena@ucsp.edu.pe

ABSTRACT

The CPU simulators has become an important aspect of teaching computer architecture since they show to the students the execution of a program inside of a particular processor. However, the majority of them are not user friendly and do not show the logic circuits inside the processor working together in order to execute an instruction. The purpose of this paper is to present the design and implementation of a CPU Simulator that meets those requirements. The simulator is based in a FPGA soft processor, a multi-cycle processor in VHDL, that has been tested on the top of SPARTAN-3E development board. The FPGA soft processor was reprogrammed in C++ because a graphical user interface (GUI) and a compiler were added to the simulator. In this fashion the simulator users could understand and learn the interaccions between the Instruction Set Architecture (ISA) of the processor with its lower levels of hardware (datapath and control unit).

Keywords: Computer architecture, CPU simulator, instruction set architecture (ISA), processor.

1. INTRODUCTION

Studying computer architecture gives to the undergraduate student fundamental ideas for understanding the design and implementation of any type of computer systems. From simple devices such as calculators to complex systems like super computers. Learning these concepts, specifically the Instruction Set Architecture (ISA) and how the digital circuits working together inside the processor, it is not an easy task. According to (Clements, 2008) the ACM and IEEE Computer Society (CS) published the computer curriculum in 1991 that consider computer architecture courses for computer science and computer engineering students. Since nothings remains static ACM and IEEE CS update the original curriculum to create CC2001 and few years later they published the CC2008.

The use of simulators is essential for teaching the computer architecture courses (Money et al., 2005). Software simulators are used to assist the students in their learning of the mostly hardware concepts (Theys and Troy, 2003). Not only simulators are the principal tools in learning computer architecture but also flash-based animations such as RaVi system (Marwedel, B. and Sirocic, B., 2004). On the other hand, (Strelzoff, A, 2007), propose the use of FPGA soft processors in teaching computer architecture. The author considers this approach a valuable skill since the ideal situation would be for students to study real working processors. Moreover, previous authors such as (Jhonson, E. and Tougaw, D. 1998) and (Arias, J, and Garcia, D. 1998) and later (Chu, Y. 2006) consider modeling processors in VHDL a valuable experience that illustrated the concepts of computer architecture courses. A survey of web resources for teaching computer architecture is presented by (Yurick, W.

and Gehringer, E. 2002). The study includes not only simulator but also resources for new instructors and resources for experienced instructors.

Nowadays, there are a lot of simulators on the web. However, the majority of them are not user friendly and their interfaces do not show the concepts explained by the professors in the classroom when they discuss the interaction between the ISA and the lower levels of hardware. The design presented in this paper combine a typical simulator with a FPGA soft processor. Furthermore, the main goal of this work is not only to reflect the behavior of a real processor but also explained it in the most didactically possible way. The rest of the paper is divided in the following sections: section 2 presents the implemented hardware architecture, and the Instruction Set Architecture (ISA). Then section 3 explains the design of the simulator. In section 4 the software engineering used to design and implement the simulator is presented. Finally, the section 5 is a reflective overview of the project and section 6 contains the conclusions and the future work.

2. DESIGN OF THE ARCHITECTURE

VHDL is a hardware description language used to implement processors (Chargo et al., 2005). The datapath and the control unit are the main parts of one processor. Those parts are implemented and tested not only in simulation but also were running programs (i.e. a program that generates the Fibonacci series) on top of a SPARTAN-3E development board. The logic circuits such as ALU, registers, multiplexors among others are well known from computer architecture textbooks such as (Hennessy, J., and Patterson, D., 1997) and (Parhami, B., 2005). The accomplishment of this task helps to understand how the hardware actually works. Since one of the goals is to build implementation close to the hardware, the VHDL code was turned into C++ code to make the same process. Both implementations in C++ and VHDL, have the same classes/components as shown in Fig. 1.

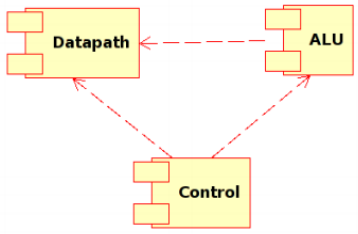

Fig. 1 Main components of the processor

2.1 MULTI-CYCLE PROCESSOR

Multi-cycle processor is designed to be able to execute a variety of instructions. The multi-cycle implementation breaks instructions down into multiple steps. Each step is designed to take one clock cycle. It allows each functional block to be used more than once time per instruction if they are used on different clock cycles. Those implementations have several keys advantages over a single cycle implementation. First to all, it can share modules, allowing the use of fewer hardware components. Secondly, instead of multiple Arithmetic Logic Units (ALUs) the multi-cycle implementation uses only one. Furthermore, only one memory is used not only for data memory but also for the instruction memory (Parhami, 2005).

In the design of multi-cycle processor (Hennessy et al., 1982) are included several registers (buffers) to hold the output of the previous clock cycle to generate the multi-cycle behavior such as Instruction Register, Memory Data Register, and ALU Register among others. The multi-cycle machine breaks instructions into a series of steps. These steps typically are the:

- Instruction fetch
- Instruction decode and Register fetch
- Execution, memory address calculation, or branch completion
- Memory access or register type instruction completion
- Memory read completion

During the first step, the multi-cycle processor reads the instructions located in the address indicate by the Program Counter (PC) and computes the address of the next instruction, by incrementing the PC. In the next step the processor decodes the instruction to determinate its type (memory access, register, immediate or branch). The third step produces different results, depending on the instruction type. For a memory access instruction, the ALU computes the memory address. In the case of a register type instruction the arithmetic is calculated. For a branch and jump instructions this is the last step where the next PC address is computed and stored. The fourth step takes place not only in register type and immediate type instructions but also in the load word (lw), store word (sw) instructions. Values are either loaded from memory and stored into the data memory register, or loaded from a register and stored back into the memory. This fourth step is the last one for register type and for immediate type instructions. For register and immediate type instructions this is the step where the result is computed by the ALU and it is stored back into the destination register. Finally, in the fifth step is executed by the load word (lw) instruction the value of the data memory register is stored back into the register file.
Different steps are all controlled and managed by the controller which is a finite state machine that uses the Operation Code (OP) to walk the rest of the components through all the different steps or states. The controller is charge of not only the time to write registers but also the operation that the ALU is performing.

The design of this processor can be broken down into two main blocks diagrams or components. The first block diagram is the datapath, and the second block diagram is the finite state machine that controls the first one. The block diagram of datapath contains all the individual modules (ALU, PC, memory, multiplexors among others) as well as the wire interconnections between them. The finite state machine has the different states, outputs, and the connections between them (Linder et al., 2005).

2.2 CONTROL

The control must specify both the signals to be set in any step and the next step in the sequence. Therefore a finite state machine is used. The decoders are used in the processor to obtain the OP and Extension Operation Code (FN) fields of each instruction. Additionally, a decoder was used to know the status of the state machine which in turn provides signal control to the datapath.

The inputs to the state machine are the OP and FN that have 6 bits each one and came from the bits of the Instruction Register. The outputs of the state machine are the control signals of the single functional units of the processor implementation especially the multiplexers of the datapath.

The execution of one instruction is broken into clock cycles which mean that each instruction is divided into a series of steps. The execution of an instruction is divided into maximal five steps. Different elements of the data path could work in parallel during one clock cycle, whereas others can only be used in series. The control signals enable different circuits in the datapath so that they can perform certain operations in one clock cycle.

2.3 DATAPATH

As shown in Fig. 2, datapath is a collection of registers, multiplexers and functional units, such as the ALU that perform data processing operations. The datapath runs on a separate thread (in C++) that allows the hardware

2.3.1 CACHE MEMORY

Cache memory is the first element needed where the instructions are stored and each instruction has a unique memory address pointed by the PC. A control signal enables the cache memory to be written, otherwise data is only read. The data bus has 32 bits. It is not possible to read and write at the same time. In order to increment the PC to the address of the next instruction the address must be kept in PC. After fetching one instruction from the cache memory, the program counter has to be incremented so that it points to the address of the next instruction during the first cycle.

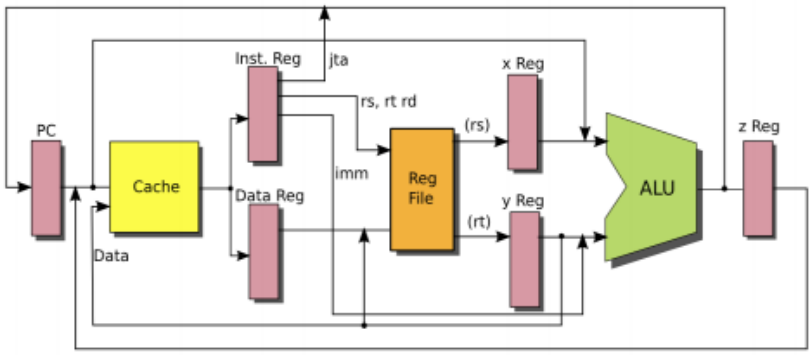

Fig. 2 Datapath

2.3.2 INSTRUCTIONS REGISTER AND MEMORY DATA REGISTER

The Instruction Register and the Memory Data Register are added to save the output that provides the cache memory. The Instructions Register decodes the instruction register where it derives the fields OP, RS (1-source register), RT(2-source register), RD (destination register), SH (amount of shift), FN, IMM (immediate operand), and JTA (jump target address).

2.3.3 REGISTER FILE

Register File is a set of 32 registers of 32 bits. The component has two address inputs, one data input, two data outputs and control lines. The inputs are 5 bits wide and specify the register number to be read, the outputs are 32 bits. To write the outcomes, two inputs are needed: one to specify the register number and one to supply the data to be written.

2.3.4 MULTIPLEXERS

The multiplexers are used in the datapath to choose between different types of input data. In this design, it has several multiplexers; one of them is a multiplexer to the cache memory entry which chooses between the address provided by PC or the address provided by Z register. Additionally, there are two multiplexers to the input File Register. The first multiplexer chooses the destination register and the second multiplexer the data to write to the file register.

In order to choose a register or the PC, a multiplexer is added for the first ALU input. The multiplexer at the second ALU input is added to choose between the constant 4 to increment the PC, a register, the sign-extended or shifted offset field for the branch instruction. Finally, the last multiplexer is connected with the PC to choose between the different input data the direction of the next instruction to execute.

2.3.5 ARITHMETIC-LOGIC UNIT

It performs basic arithmetic, shift and logic operations which are controlled by the OP and FN. The result of the instruction is written to the output. Additionally, the ALU has a zero-bit indicator that became active when the result of the last operation was equals to zero; also there is an overflow-bit that indicates when the operation is invalidating (overflow).

All the instructions we use read two registers, perform an ALU operation and write back the result. To perform basic arithmetic, shift and logic operations, as the hardware do, it runs every possible way but in the end the result will depend on the final multiplexer.

2.4 INSTRUCTION FORMAT

The format of the instruction used in the simulator is shown in Fig. 3.

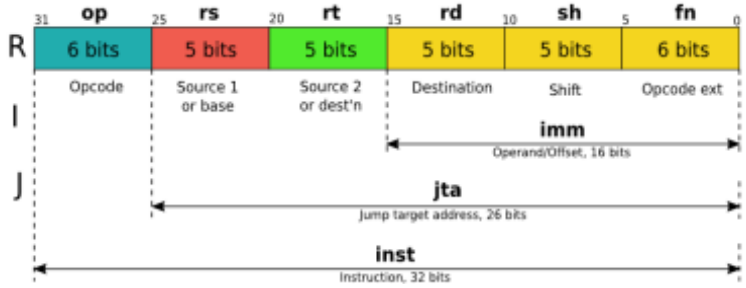

Fig. 3 Format of the instruction

3. DESIGN OF THE SIMULATION

The main processor components (hardware) are considered in the design of the simulator as shown in Fig. 4. Beside hardware, a compiler is implemented and also a handler for interactions between GUI, Hardware and Compiler. The simulator was implemented in C++ by using Qt libraries.

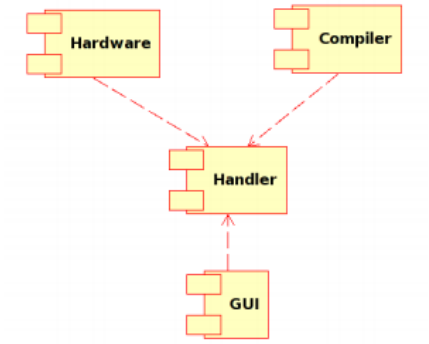

Fig. 4 Main components of the simulator

3.1 COMPILER

The compiler was implemented to handle the grammar in a dynamic way. To achieve this characteristic a Patricia Trie was used but this feature does not change the logic of the compiler (Fig. 5).

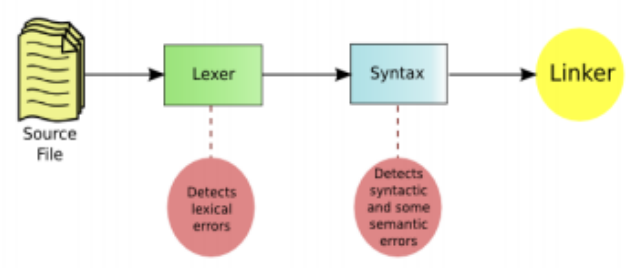

Fig. 5 Compiler

3.1.1 PATRICIA TRIE

A Patricia Trie is implemented to handle the grammar dynamically (Gonnet, G. 1984). A Patricia Trie is chosen instead of another data structure because it provides fast string searching in an optimal space. The grammar was then coded and stored in the Patricia Trie and then the instructions were allowed to be inserted. In this fashion, user defined grammars and pseudo instructions are handled in an efficient way.

The functions implemented to support such behavior were: lookup, insert, delete, find predecessor and find successor. Each node stores whether it was a terminal node or not and information about the instruction represents.

3.1.2 GRAMMAR FILE

The Patricia Trie requires a file that contains the instruction set architecture, called grammar file. In the grammar file # is a comment. The strings that start with “T_” are key words only for the grammar file for example.

- String starts with T_REG can have any register.
- T_REG_S is a 5-bit and has a specific location on the 32-bit instruction, see 4.
- T_IMM_VALUE is a isa 16-bit and has a specific location on the32-bit instruction, see4.
- The other strings like "+", "IN" are the keywords that you can use to write an instruction, for example:
- T_REG_S + T_REG_T IN T_REG_D means that you can put any register, @4+@5 in @6 you are adding the value of the register 4 and the register 5 and stored in register 6.
- The final string that starts with "0x" is the 32-bit decode instruction, for example:
- T_REG_S + T_IMM_VALUEIN T_REG_T 0x00100000000000000000000000000000 32-bit decode.
- T_REG_S + T REG_T IN T_REG_D 0x00100000 this instruction does not have 32-bit but we can write the same code in 32-bits 0x00000000000000000000000000100000.

3.1.3 LEXER

The main objective of the lexer is to get a token-based instruction; registers are symbolized with initial @, for example.

- @1 means the first register.
- @32 is not valid because the registers are from 0 to 31.

Fig. 6 presents the lexer. It provides us the Token number or an error.

```
/**
  Classifies the code by tokens.
  @returns int, with the current token index.
*/
int CLexer::lex()
{
    ...
    // a register always starts with @
    else if(m_current == QLatin1Char('@'))
    {
        m_reg = 0;
        while (next().isDigit())


     {
         m_reg *= 10;
         m_reg += m_current.digitValue();
     }
     // register from 0 to 31
     if(m_reg >= 0 && m_reg <= 31)
         return m_tokensTable.value("t_reg"); //
            T_REG;
     else
         m_errorMsg = "wrong register number " +
            QString::number(m_reg);
 }
 ...
 return ERROR;
}
```

Fig. 6 Lexer

3.1.4 SYNTAX

The syntax has two main objectives, loads the grammar from a file and finds syntactic errors. The grammar is encoded before being inserted. Fig. 7 shows the insert operation in the Patrice Trie, to encode the grammar a number is assigned to each token in the grammar file. Then the objects inserted in the Patricia Trie are strings like "35421".

```
/**
  Adds an intruction to the grammar.
  @param[in] _inTokens, QStringList represents the
     instruction.
  @param[in] _inColor, int differs the instructions
     and pseudoinstructions. for default instructions
     with 0.
  @returns bool, true, if the instruction has been
     loaded correctly, false otherwise.
*/
bool CSyntax::addGrammar(QStringList _inTokens, int
   _inColor){
    QString toPatty;
    QString hexValue = QString();
    QStack<QString> stack;
    for ( QStringList::Iterator it = _inTokens.begin()
       ; it != _inTokens.end(); ++it ) {
        QString token = *it;
        if(!token.startsWith("0x"))
        {
            if(token.startsWith("t_reg"))
            {
                token = "t_reg";
                stack.push(*it);
            }
            if(!m_ptrLexer->contains(token))

                    m_ptrLexer->insert(token, ++
                       m_tokensIndex, _inColor);
                toPatty.append(QString::number(m_ptrLexer
                   ->value(token)) + " ");
            }else
                hexValue = token;
        }
        hexValue = hexValue.right(hexValue.size() - 2);
        CInstruction inst(hexValue,stack);
        return m_ptrPatty->insert(toPatty, new QVariant(
           m_instIndex,&inst));
    }
```

Fig. 7 Insert operations in Patricia Tri

3.1.5 LINKER

The main goal of the linker is to decode the instruction, for instance:

- T_REG_S + T_IMM_VALUE IN T_REG_T 0x00100000000000000000000000000000 this instruction is intended to add a register with a immediate value, @5 + -15 IN @7, so the linker would do 0x00100000101001111111111111110001 (the immediate value can be negative too), @5 + 15 IN @7 0x00100000101001110000000000001111.

So this is how you can create your own ISA, example (do this task in an automatically way would be an improvement).

- T_REG_S + T_IMM_VALUE IN T_REG_T 0x00100000000000000000000000000000 this instruction is intended to add two registers.
- T_REG_S plus T_REG_T IN T_REG_D 0x00100000 this instruction is intended to add two registers, however plus is using instead of + in order to produce two difference instructions.
- T_REG_D = T_REG_S + T_REG_T 0x00100000 this instruction is also intended to add two registers, in a more familiar way. To apply this approach it is necessary to take care about the registers order.

And finally you can create your on pseudo-instructions, for example: Do this task in an automatically way would another improvement.

3.2 GRAPHICAL USER INTERFACE (GUI)

Qt libraries were chosen to work the user interface because they offer multiplatform support with out efficiency problems, since Qt is implemented in C++. The GUI is designed in a way that every component is independent from each other and independent from the hardware and compiler. The GUI components are shown in Fig. 8.

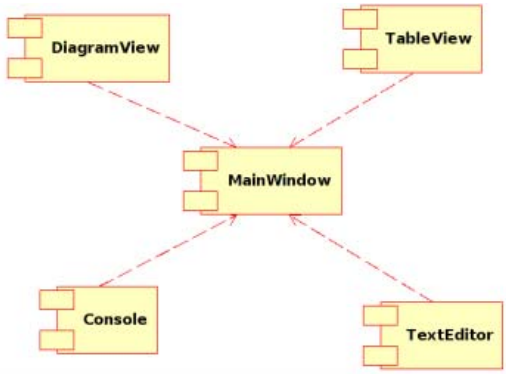

Fig. 8 Components of the GUI

3.2.1 MAIN WINDOW

The main window is the user interface component that manages the other component switch which the user interacts. These components are:

- MenuBar containing File, Edit, Execution, Grammar and Help Menus (Fig. 9.1)
- Toolbars file, edit toolbars (Fig. 9.2) and execution toolbar (Fig. 9.3). Execution Toolbar Allows compiling, running, running by instruction, running by step and stopping the current program.
- CentralWidget A Multiple Document Interface (MDI) is used as the central widget, this way allowing to work in multiple programs codes. (Fig. 9.9)
- Dockwidgets that store the directories, register, cache, memory, ALU, datapath and control views (figures. 9.4, 9.5, 9.6 and 9.8).

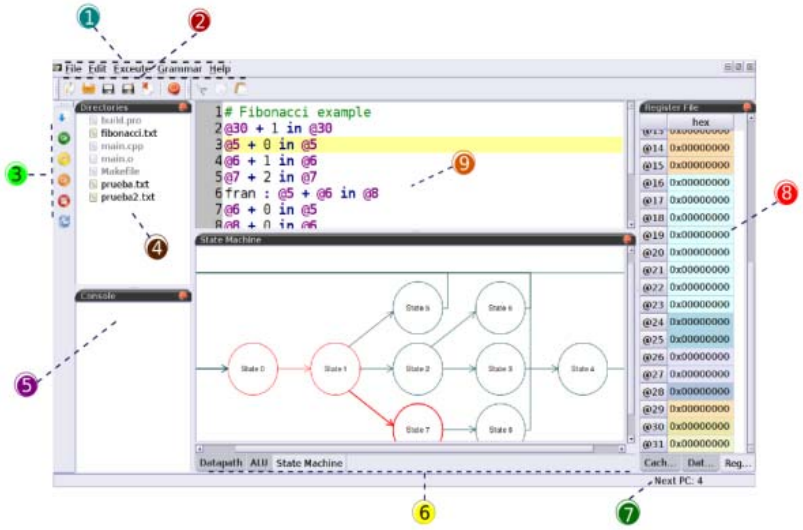

Fig. 9 GUI

3.2.2 DIAGRAM VIEW

This component (Fig. 9.6) manages ALU, datapath and control unit views. It manages hardware components as nodes and buses as links. The editor handler component manages the interaction between each hardware component allowing the animation of each view reflecting how hardware is actually working.

3.2.3 TEXT EDITOR

Each text editor in the MDI allows the users to view and edit the code of the program. The text editor also provides auto complete and highlight coding features.

3.2.4 CONSOLE

Two consoles were used in the simulator. The first one is for the simulator itself, meanwhile the other shows if there is an error and indicates the error type and the number of the line.

4. USE OF SOFTWARE ENGINEERING IN CREATING THE SIMULATOR

To develop the simulator, an initial high level planning stage is carried out, the goal of this stage is to establish the development process and best practices to be used. In this stage the scope of the project was defined, including objectives, the project team organization and outline the deliverables. With the needs in mind the iterative model with short cycles accompanied and test driven development for some critical classes was chosen.

To start the cycles, the project was divided into modules, and each module into activities small enough to have short cycles with visible results. Then, a priority was assigned to start with the highest priority activities. Each cycle includes the following phases: planning, analysis and design, implementation and testing.

Unit tests were written before start the implementation of some classes. Before finishing each cycle the code was tested in Debian GNU/Linux and Windows. In that fashion bugs were fixed. We have also set unit tests for the main classes.

A simple coding standard for both, VHDL and C++ (names classes, data members, and function arguments among others) is defined. The authors follow java doc standard for documenting the code, define methods of communication as google groups, the use of SVN to work in the C++ code, the use of Model-View-Controller architectural pattern.

5. REFLECTIVE OVERVIEW

To develop the simulator the contribution of people from different areas was needed. Our goal is to design a user friendly simulator with educational purposes showing the behavior of a CPU. We decide that the CPU implementation must be close to reality (the core is a FPGA soft processor), carrying every instruction and component to digital circuit (bits level). This is the main reason to implement the hardware architecture in VHDL and then we follow the same logic in C++. Probably this approach makes the implementation harder and takes more time however this decision could lead to a better and more robust result.

Furthermore, thought that the best option is to implement a multi-cycle processor because it could have access to more relevant information that can be displayed in the simulator. The data structure like a graph or a tree to hold the grammar but Patricia Trie is best election since it allows fast searching and inserting grammar rules dynamically, in that way the user could add pseudo instructions. The use of Qt libraries is because it allows code to be ported to different platforms, this feature was important because the development was done in Debian GNU/Linux and we pretended to take it to Windows.

6. CONCLUSIONS AND FUTURE WORK

Implementing a FPGA soft processor in VHDL and then turn in to CPU simulator in C++ is possible. The simulator GUI allows the users to have a sense of what happened inside the CPU even through at the hardware level. The ISA is reconfigurable through the use of Patricia Trie and the simulator is able to run in GNU/Linux and Windows. Of course there are improvements to do such as:

- Allow users to add their own pseudo instructions, for this purpose we want to make a pseudo instructions editor. This editor will also allow creating your own ISA.

- Create different types of architecture (single-cycle, pipeline) and join them to the simulator would be possible by using an adapter class. So, the user could choose the architecture on which he wants to run his program.
- Since we already have the VHDL implementation running on a Spartan 3E development board, the next step would be to join it with the simulator implementing an adapter. It means turn in the simulator into emulator.

REFERENCES

Arias, J, and Garcia, D. (1998), "Introducing computer architecture education in the first course of computer science career", Proc of Workshop on Computer architecture, WCAE.

Chargo J. and Falat M. (2005). "Multicycle Processor Design in Verilog", *ISU*.

Clements, A. (2008), "Crafting a Curriculum in Computer Architecture", *Proc of 38th annual Frontiers in Education*, Vol. 2, pp T3E-15 – T3E-20.

Chu, Y. (2006). "A Simple Project Paradigm for Teaching Computer Architecture", *Computers in Education Journal, Computers in Education Division of ASEE* VOL. XVI No.2, pp. 106-112,

Gonnet, G.(1984), "Handbook of Algorithms and Data Structures". *Addison-Wesley, International Computer Series,* pp 109.

Jhonson, E. and Tougaw, D. (1998), "An integrated Computer Architecture Experience", *Proc of 28th annual Frontiers in Education*, Vol. 3, pp 1208-1211.

Marwedel, B. and Sirocic, B. (2004), "Bridges to computer architecture education", *Proc Workshop on Computer Architecture Education held in conjunction with the 31st International Symposium on Computer Architecture*, Vol. 2, pp T3E-15 – T3E-20.

Money, D., and Harris, S. (2007). "Digital design and computer architecture", *New York:Elsevier Inc.*

Parhami, B. (2005). "Computer Architecture: From Microprocessors to Supercomputers", *Oxford University Press.*

Linder M. and Schmid M. (2007). "Processor Implementation in VHDL", *Master Thesis, University of Ulster at Jordanstown.*

Hennessy, J., Jouppi, N., Przybylski, S., C.Rowen, Gross, T., Baskett, F. and Gill, J. (1982). "MIPS: A microprocessor Architecture*", Proc of the 15th annual workshop on Microprogramming, pp.17-22*.

Hennessy, J., and Patterson, D. (1997), "Computer Architecture: A Quantitative Approach*", Morgan Kaufman.*

Strelzoff, A (2007), "Teaching Computer Architecture with FPGA Soft Processors", *Proc of ASEE SE Conference* Vol 1, 2.43.

Theys, M. and Troy, P. (2003). "Lessons learned from teaching computer architecture to computer science students". *Proc of 33rd annual Frontiers in Education*, Vol. 2, pp 7-12.

Yurick, W. and Gehringer, E. (2002). "A survey of Web Resources for Teaching Computer Architecture". *Proc Workshop on Computer Architecture Education held in conjunction with the 29th International Symposium on Computer Architecture*, Article No. 23.

Authorization and Disclaimer

Authors authorize LACCEI to publish the paper in the conference proceedings. Neither LACCEI nor the editors are responsible either for the content or for the implications of what is expressed in the paper.