# Building a microprocessor architecture at Computer Engineering undergraduate courses

Morgana Sartor, Thaynara Tessaline Mitie Sei Soares, Marcelo Daniel Berejuck

Department of Computer Engineering, Federal University of Santa Catarina – UFSC - Brazil

***Abstract***— *The learning process of Computer Organization and Architecture is fundamental for students of Computer Engineering and related areas. The complexity and lack of familiarity of the students with the content make it difficult to understand essential concepts for the development of fundamental skills for a computing area professional. This paper introduces a conceptual microprocessor implemented in three different courses of Computing Engineering undergraduate: Computer Organization and Architecture, Very Large-Scale Integration (VLSI) Circuit Design, and Embedded Systems Design. The overall idea is to link the fundamental concepts of Computer Organization and Architecture with hands-on opportunity to develop the blocks, such as registers or program memory, using a hardware description language, and applying this microprocessor on embedded systems design, using it as a softcore processor. Initial results showed the students get more involved in their learning process when they understand the usage and application of those concepts.*

***Keywords*— *Computer Organization and Architecture, Embedded Systems Design, Learning Process, Very Large-Scale Integration.***

## I. INTRODUCTION

The design of an embedded system implies to choose a suitable microprocessor to achieve the features of the design in terms of power consumption, raw material cost, or memory access capacity, for instance. It means that it is essential for Computer Engineering students to understand the architecture and organization of microprocessors to design an embedded system in their future professional career properly.

We noticed that some concepts addressed in Programming Methodology and Algorithms are well understood by the students when they have the opportunity to associate these concepts with the internal organization of the microprocessor they are dealing with. Didactic microprocessors simulators have been proposed to improve the learning process. Most of them have a simplified architecture in terms of machine language, and a limited organization in terms of external interfaces, such as input and output pins, and other interfaces. These simulators are usually software applications written with high-level languages, like Java or C#. Most of them do not allow the users to make changes on them, nor integrate them with other systems, for example, a physical or synthesizable microprocessor.

In this paper, we are suggesting the adoption of a software tool to fulfil this gap. The tool is an application that simulates a simplified microprocessor called µPD. At this point, the tool is similar to the other ones proposed so far and is suitable for the Computer Organization and Architecture course. The advantage of this tool is its capability to generate VHDL code (acronym of Very high-speed integrated circuit Hardware Description Language). The tool generates the internal structure of the µPD processor as components (or blocks), like registers block or memory block. The students of the VLSI Circuit Design course can connect those blocks to build an entire µPD microprocessor and check its functionalities. It is the second opportunity that the students have to work with the microprocessor.

The third opportunity is in the Embedded Systems Design course. At this point, the students already have a softcore well tested and working, and in this course, the students plan and implement embedded systems based on this softcore. The students must do new VHDL components according to their design needs.

We organized this document in sections. Section two introduces the related work. Section three was divided into subsections and introduced the Organization and Architecture of µPD processor, in terms of internal blocks, Instruction Set, for instance. It also introduces the two simulators developed: Text simulator and Block simulator. Section four introduces the initial evaluation done with undergraduate students of Computer Engineering. We finished this paper with some conclusions and target for future work.

## II. RELATED WORK

This section introduces the main academic simulators adopted as engineering education tools. It was divided into subsections, in which the main features of those tools are outlined. At the end of this section, we introduce a table outlining the µPD tool with the other ones previously described.

### MARS

The MARS simulator was developed by [1]. It is an IDE (Integrated Development Environment) that simulates a microprocessor MIPS. Several engineering colleges adopted MIPS tool for practical experiments due to its simplicity and regularity. In other words, it has 32 registers, and each of them with 32 bits [2]. MARS has an integrated editor, featuring multiple file-editing tabs, with context-sensitive input, and colour-coded assembly syntax. All assembly files in a folder may be assembled into a single executable. It also has a command-line mode for tests and to evaluate many programs. The command-line arguments allow specifying registers and memory locations to be displayed after the program run to examine for correct contents. A differential feature of MARS is an option called "Tool" utility for MIPS control of simulated devices. The authors define a Tool utility as a program running on a separate thread with access to MARS data. An assembly program can run in MARS and interact with the tool through memory-mapped IO. Due to this feature, several pseudo-devices can be interfaced to MIPS assembly code, or extended to physical devices or hardware.

### QtSPIM

QtSPIM, available at [3], is a self-contained simulator that runs programs for MIP32. It works as a “command terminal”, and the users can read and execute assembly language programs written for MIPS32 processor. QtSPIM also provides a simple debugger and minimal set of operating system services. It is a tool that implements much MIPS32 assembler-extended instruction set. Some exceptions are the most floating-point comparisons, some rounding modes, and the memory system page tables. The MIPS architecture has several variants that differ in various ways (e.g., the MIPS64 architecture supports 64-bit integers and addresses), which means that QtSPIM will not run programs for all MIPS processors. As weakness, it does not have an IDE well elaborated with breakpoints, and other useful debug features and is not possible to execute a program step-by-step.

### CPUlator

CPUlator, also know as “Computer System Simulator”[4] is a tool capable of simulating three architectures: Nios II, ARMv7, and MIPS. Its simulation is done as a computer system, e.g. a processor and I/O devices. To prevent issues related to the installation, those authors have built the debugger that runs in a regular web browser. It was designed as a tool for learning assembly-language programming and computer organization. As main features that may be highlighted are the debug options offered by the tool: Single-step, breakpoints, watchpoints, trace, call stack, examine disassembly, memory and registers. It’s the tool with more debug features we found so far. As inputs, it accepts both assembly source code and ELF executables.

### ARMSim#

ARMSim# [5] was conceived as a desktop application developed to run in a Windows environment (.NET3.0). It allows users to simulate the execution of ARM assembly language programs on a system based on the ARM7TDMI processor, together with collecting statistics for execution and cache usage [5]. It also can be extensible through plugins, which provide an interactive environment for I/O by simulating examples of embedded systems. The authors have implemented several plugins, and they are available for download. An important feature is the addition of graphical views, programmed as plugins, allowing execution of interactive interfaces and supporting interrupt processing. The main view included is that of a board including an ARM processor and a set of peripherals - buttons, LED lights, keyboard, small LCD screen.

### CompSim

CompSim is an IDE developed by [6] to be an assistance tool for Computer Organization and Architecture course. It uses a proprietary 16 bits processor called Cariri: an accumulator-based processor with 16 instructions set. The development was done using Python v3.5 and can be run on MS Windows and GNU Linux (32 and 64 bits). The IDE has visual components that emulate the hardware, like memory and input / output pins. Some of this hardware is configurable like the RAM memory, for instance. This memory can be configured as Direct Mapped, Fully Associative, and Set Associative.

### SimusS

SimusS was proposed by [7], and it is based on a configurable processor, called Neander-X. The processor has a variable instruction size, from 1 up to 3 bytes, four addressing types and 31 assembly instruction. SimusS has a simple IDE interface emulating input and output devices. The IDE also has a text editor and an assembler integrated. It is possible to visualize the internal data path of the processor during the debug action, further the external input and output interfaces.

### Bipide

Bipide was proposed by [8] as an integrated environment in which it is possible to develop an algorithm and execute in a virtual 16 bits processor, called BIP. Different from the Neander-X, the BIP processor has a regular instruction size with 16 bits. The main difference is the natural language used as the programming language. BIP was developed to be used on early stages of Computing Science and Computing Engineering when the students are taking the first algorithm programming course. The students can check the data path inside the processor graphically while the algorithm is executed step-by-step.

### CESAR and RAMSES

CESAR was developed by [9] and, on the same way of the other tools we mentioned in this section, it also focuses on Computer Organization and Architecture learning. The proprietary processor has 16 bits and 41 assembly instructions. The graphical environment of this tool is simple, showing register related to the ROM memory, register related to the RAM memory, and input and output pins. The main difference of this tool to the other ones is the data representation, based on 2's complement.

RAMSES processor is an 8 bits version of CESAR processor. It has four addressing modes, while the CESAR has eight. The instruction set is also different; for the RAMSES, there are only 16 assembly instructions. The graphical interface is quite similar to the CESAR interface, only changing the registers sizes.

### COCONUT

Authors [10] proposed a visual educational configurable simulator for computer architecture and organization, called COCONUT. According to them, the students can create their own processor with an arbitrary architecture and simple organization on the register transfer level, write an assembly program to be executed on the processor and observe the instruction execution phases of the program on the processor. The students create a VHDL processor using some basic blocks and following some rules do connect them. A simulator is available to evaluate and test the implemented processor. The configurable part allows students to use the simulator for defining the instruction decoding of the processor operation unit and the content of the microprogram memory of the processor control unit.

### EduMIPS64

Authors [11] proposed the visual EduMIPS64 CPU simulator as supporting to undergraduate computer architecture students. It implements a 5-stages pipeline, allowing instruction-level parallelism. As each virtual CPU cycle is executed, the user interface updates all its components to display the representation of the current program execution state. The simulator was written in Java language and its graphical interface implements, further the necessary registers and memories, some statistics on the execution, such as the number of Cycles Per Instruction and the number and class of stalls.

### MarieSim

Authors [12] developed the MarieSim: a computer architecture simulator based on the MARIE architecture and designed to teach beginning computer organization and architecture. It has a graphical environment that allows students to observe how assembly language statements affect the registers and memory of a computer system. The graphical environment for MarieSim and the accompanying data path simulator was written in Java Swing, and the integrated MARIE assembler was written in Java. MARIE is the acronym of "Machine Architecture Really Intuitive and Easy"

## Overview of related work

This Subsection summarizes the academic simulators reviewed in Section with Table 1 outlining the main features of each simulator. µPD features were also included at the end of the table. Note that µPD is the unique toolset that generates VHDL code. The code generated is used by the students on the other two courses: VLSI Circuit Design, and Embedded Systems Design.

*Table.1: Overview of the academic simulators introduced above. Grey colour means the simulator has that feature.*

| | MARS | QtSPIM | CPUlator | ARMSim# | COCONUT | EduMIPS64 | MarieSim | CompSim | SimmuS | Bipide | Cesar | µPD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No installation required | | ▨ | | | | | | | | | | |
| Editor | ▨ | | | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| Assembler | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | Natural language | ▨ | ▨ |
| Debugger | ▨ | Few options | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| Block simulation | | | ▨ | | | | | | | ▨ | | ▨ |
| I/O interface | | | | | | | | | ▨ | | ▨ | ▨ |
| Interrupt interface | ▨ | | | | | | | | | | ▨ | ▨ |
| Depurator | | | | | | | | | ▨ | | | |
| Data-path visible | | | | | | | ▨ | | | ▨ | | ▨ |
| Based on Accumulator | | | | | | | ▨ | ▨ | ▨ | ▨ | | |
| Based on Registers | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | | | | | ▨ | ▨ |
| Generate VHDL code | | | | | ▨ | | | | | | | ▨ |

## III. µPD SOFT PROCESSOR

Over the past three decades, the Association for Computing Machinery (ACM) and the Computer Society of the Institute for Electrical and Electronics Engineers (IEEE-CS) joined efforts to promote and upgrade a document periodically with curricular guidelines called "Computing Curricula" [14],[15]. These guidelines are defining broad knowledge areas that apply to all computer engineering programs.

According [16], the six knowledge units with five or more core hours in the Computing Curricula are: Fundamentals of computer architecture, Memory system organization and architecture, Interfacing and communication, Device subsystems, Processor systems design, and Organization of the CPU.

The three courses we are working (Computer Organization and Architecture, Very Large-Scale Integration – VLSI - Circuit Design, and Embedded Systems Design) deal with this knowledge in different manners. So, we intended to propose a softcore simulator capable of being used by these courses and covering the knowledge units with varying levels of difficulty.

### Platform Overview

According to the authors [13], a platform consists of a library of components associated with its composition rules. These standards allow the reuse of cores and the communication infrastructure, thus creating the concept of a development platform. We understand that we have to work with this target in mind: elaborate a platform based on the software processor µPD. So far, we have developed some software tools: editor, assembler, software simulation, and VHDL code generator, all of them forming a toolset. Furthermore, the VHDL code generated by one of these tools has a standard interface and connection rules, which allows the students to build large embedded systems. The standard interface and connection rules are on the way to build a future platform.

### Processor Organization

The didactic processor we are proposing is a monocycle RISC (Reduced Instruction Set Computer) with 16 bits data bus. It connects to an external ROM (Read Ony Memory), that contains the machine code given by the Assembler tool. To connect to other devices (or external components), it has an input bus and an output bus, as depicted in Figure 1. The signals INT are six external interrupts supported by the processor, ADDR is an address bus, and DIN and DOUT are input data and output data, respectively.

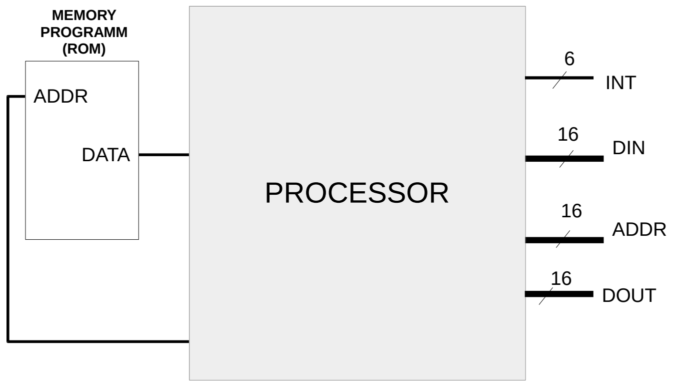


*Fig. 1: ROM memory, processor and its connection bus.*

Figure 2 illustrates the internal processor blocks. It was divided into two logic areas called “data path”, bordered by a blue dashed line and “control path”, bordered by a pink dashed line. The control path is responsible for the reading of instructions stored in the ROM memory, and to manage the execution of data flow. This management is done by the “Control block” show in Figure 2. Another block that belongs to the control path is the “LIFO memory”, acronym of “Last In, First Out”. When a subroutine is called, or an interrupt occurs, the control block saves the content of the registers (Register block in the data path) before it jumps to the new program memory address.

The data path has six blocks: a block of registers, an arithmetic and logic unit (ALU), a data memory (RAM type) and three single registers. The block of registers implements four registers for general purpose. We have chosen a small number of registers to enforce the students to manage data between the data memory and the block of registers (context saving). The ALU implements two mathematical operations (addition and subtraction), four logical operations (AND, OR, XOR, and NOT), and three jumps: jump if two registers have the same value (JE), jump is a register is zero (JZ), and unconditionally jump (JI). The three single registers are used in the data path to store temporary data for the external bus (output) and from the external bus (input).

The internal data memory (called as RAM) has ten address bits, which allows it to store up to 1K words (16 bits each). The processor can access up to 4K in program memory (called ROM). This quantity of memory is shared by the main program and by the six interrupts allowed by the µPD processor (INT0 up to INT5), and Table 2 depicts these memory ranges. Each instruction in the program memory has 17 bits, and these instructions will be detailed in the next subsection.

*Table.2: Program memory address range.*

| Area | Begin (hex) | End (hex) |
|---|---|---|
| Main programm | 000 | 3FF |
| INT 0 | 400 | 4FF |
| INT 1 | 500 | 5FF |
| INT 2 | 600 | 6FF |
| INT 3 | 700 | 7FF |
| INT 4 | 800 | 8FF |
| INT 5 | 900 | 9FF |

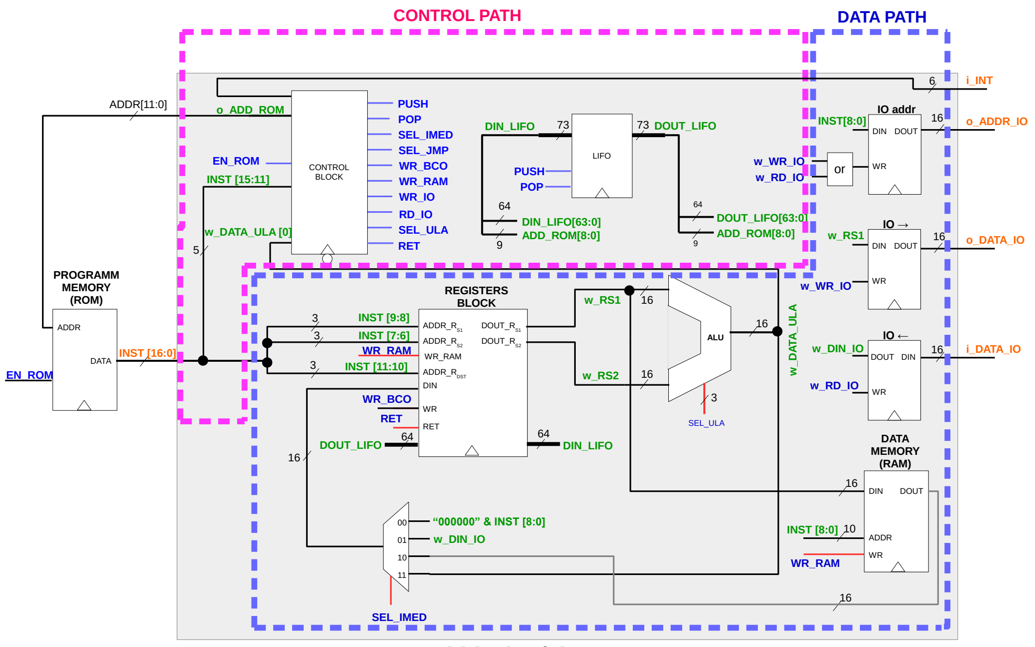


*Fig. 2: Internal block of the µPD processor.*

## Processor Architecture

The µPD architecture is based on registers. There are four registers for general purpose, called R0, R1, R2, and R3, all of them inside the Register block (Figure 2). To deal with registers were implemented 21 opcodes in the Instruction Set Architecture (ISA), as shown in Table 3. Int that table, RS1, RS2, and RDST can assume one of the four purpose registers, R0 up to R3.

All instructions stored into the program memory have 17 bits, and they are organized as fields as shown in Figure 3. The most significant five bits are using as opcode of instructions. The fields RS1 and RS2 are called "source registers", and have two bits each. The ALU will treat the values stored into these registers. On the same way, RDST is the register in which the ALU will store the result of operation done with the register RS1 and RS2.

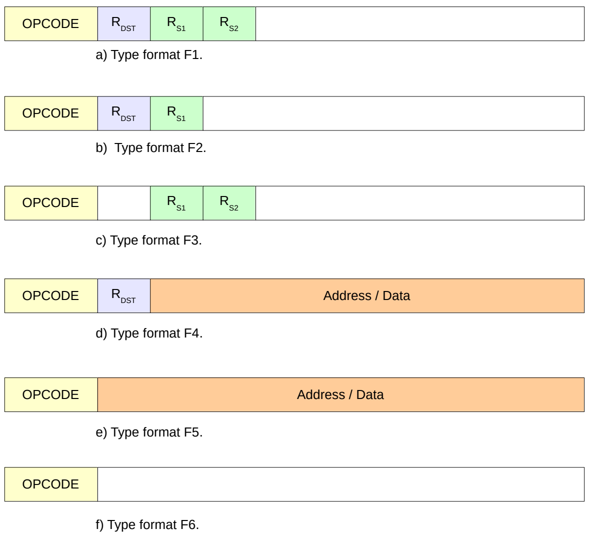


*Fig. 3: Instructions format.*

As depicted in Figure 3, there are six instructions formats, and we called them as F1 up to F6. These instructions are described in Table 3, and the first column informs which format is related to each opcode.

*Table.3: µPD Instruction Set Architecture.*

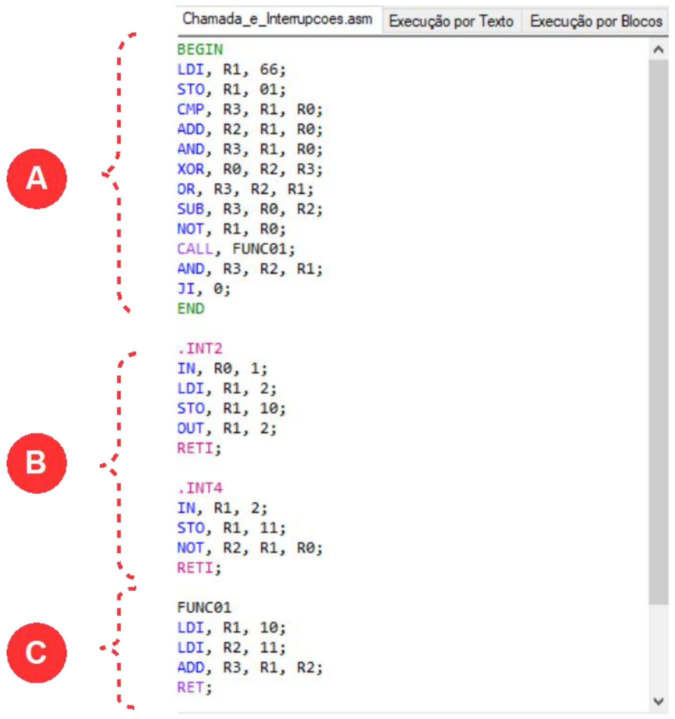
| Type | Mnemonic | Opcode | Descripton |
|---|---|---|---|
| **F1** | ADD | 00010 | Add: $R_{DST} = R_{S1} + R_{S2}$. |
| | SUB | 00011 | Sub: $R_{DST} = R_{S1} – R_{S2}$. |
| **F4** | IN | 00101 | External bus Input: $R_{S1}$ |
| | OUT | 00100 | External bus output: $R_{S1}$. |
| | LD | 00111 | Load register with data from the memory: $R_{S1}$ |
| | LDI | 00001 | Load register with immediate value: $R_{S1}$. |
| | STO | 01000 | Store content of register into the memory: $R_{S1}$. |
| | JE | 01010 | Jump if two register have the same value. |
| | JZ | 01001 | Jump if the register is zero. |
| **F5** | JI | 00110 | Jump unconditional. |
| **F3** | CMP | 10100 | Compare value of two registers. If they are equal, set 1. |
| **F1** | AND | 01011 | Logical AND between registers: $R_{S1}$, $R_{S2}$. |
| | OR | 01100 | Logical OR between registers: $R_{S1}$, $R_{S2}$. |
| | XOR | 01101 | Logical XOR between registers: $R_{S1}$, $R_{S2}$. |
| **F2** | NOT | 01110 | Logical NOT over a register: $R_{S1}$. |
| **F5** | CALL | 01111 | Call subroutine. |
| **F5** | RET | 10000 | Return from subroutine . |
| | RETI | 10011 | Return from an external interrupt treatment. |
| | STOP | 10010 | Stops the processor. |
| | NOP | 00000 | No operation. |
| | SETR | 10001 | Configure which interrupts will be active. |

Given the restricted number of general-purpose registers, the students are enforced to do data transfer between these registers and the data memory (RAM), and the same happens to I/O reads and writes. The instruction IN and OUT are related to data input and data output of external I/O devices. Note that in Figure 2 there is a register (called “IO address”) with 16 bits. It means that the µPD processor can access up to 216 external devices, theoretically. The quantity of devices depends on the FPGA size and the devices (components) size, for instance.

## Graphic interface

The µPD was developed focused on interdisciplinary teaching. The students begin with the basic concepts of Organization and Architecture of Computers when they learn about the µPD and perform basics assembly language programs. On a second course, VLSI circuits design, they build the µPD processor using VHDL language and can build different devices to connect the external bus of the processor. At this point, the students have tested components and can do an embedded design based on the soft processor for FPGA.

The µPD toolset has an Editor, an assembly compiler, a simulator, and a VHDL components builder, and their main features will be introduced in this section. All tools were written using C# language and .NET framework. The first tool is the Code Editor. It has a simplified visual interface that has a different colour for reserved words. It also has some editor's features, such as line counters, cut, copy, paste, block selection, and block comment. An example of an assembly code written with the integrated Editor is shown in Figure 4.

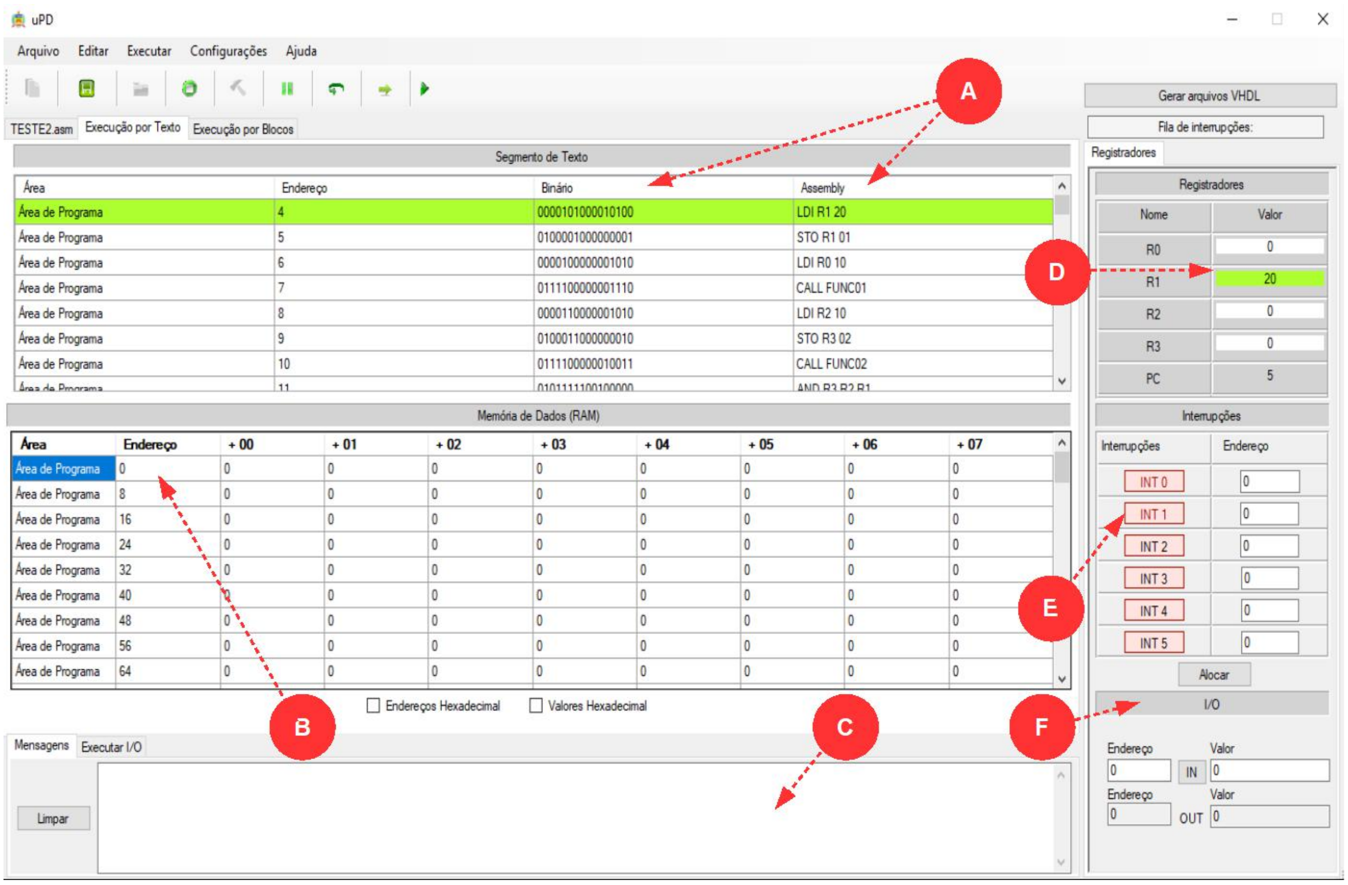


```
BEGIN
LDI, R1, 66;
STO, R1, 01;
CMP, R3, R1, R0;
ADD, R2, R1, R0;
AND, R3, R1, R0;
XOR, R0, R2, R3;
OR, R3, R2, R1;
SUB, R3, R0, R2;
NOT, R1, R0;
CALL, FUNC01;
AND, R3, R2, R1;
JI, 0;
END

.INT2
IN, R0, 1;
LDI, R1, 2;
STO, R1, 10;
OUT, R1, 2;
RETI;

.INT4
IN, R1, 2;
STO, R1, 11;
NOT, R2, R1, R0;
RETI;

FUNC01
LDI, R1, 10;
LDI, R2, 11;
ADD, R3, R1, R2;
RET;
```

*Fig. 4: Example of an assembly code written with the integrated Editor.*

Figure 4 depicts an assembly code written with the integrated Editor. Reserved words, such as LDI and NOT, was written using a blue colour. Other reserved words, such as .INT2 and RETI were written using magenta colour. The begin of the written code (indicated by a red circle with the letter A) is an example of the main program. The code selected by the red circle with the letter B is related to the code written for two interrupts (.INT2 and .INT4). On the bottom of Figure 4, the letter C written inside a red circle indicates a function code (called as FUNC01 in that example) or subroutine.

Once the code was written and assembled by the integrated Assembler tool, it can be executed in two different simulations: the "Text simulation", and "Block simulation". The Text simulation is shown in Figure 5. The red circle with a letter A is pointing to binary code (machine code) and to the assembly program. The line in green colour indicates the step of execution at that moment. In that case, the instruction is LDI R1 20, which means "load register R1 with the number 20". Note that on the right side the register R1 received the number 20 and it is green at this moment (letter D on the red circle). Letter B is pointing to the data area, also called RAM. Letter C is pointing to a window console. This console is used to display messages to the user related to the simulation under progress. Letter E shows the interrupt buttons that the user can press to simulate as an external interrupt. Letter F is the interface where the user chooses an address for the external device and can read/write data from/to the external device.



Segmento de Texto

| Área | Endereço | Binário | Assembly |
|---|---|---|---|
| Área de Programa | 4 | 0000101000010100 | LDI R1 20 |
| Área de Programa | 5 | 0100001000000001 | STO R1 01 |
| Área de Programa | 6 | 0000100000001010 | LDI R0 10 |
| Área de Programa | 7 | 0111100000001110 | CALL FUNC01 |
| Área de Programa | 8 | 0000110000001010 | LDI R2 10 |
| Área de Programa | 9 | 0100011000000010 | STO R3 02 |
| Área de Programa | 10 | 0111100000010011 | CALL FUNC02 |
| Área de Programa | 11 | 0101111100100000 | AND R3 R2 R1 |

Memória de Dados (RAM)

| Área | Endereço | + 00 | + 01 | + 02 | + 03 | + 04 | + 05 | + 06 | + 07 |
|---|---|---|---|---|---|---|---|---|---|
| Área de Programa | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Área de Programa | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Área de Programa | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Área de Programa | 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Área de Programa | 32 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Área de Programa | 40 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Área de Programa | 48 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Área de Programa | 56 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Área de Programa | 64 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Registradores

| Nome | Valor |
|---|---|
| R0 | 0 |
| R1 | 20 |
| R2 | 0 |
| R3 | 0 |
| PC | 5 |

Interrupções

| Interrupções | Endereço |
|---|---|
| INT 0 | 0 |
| INT 1 | 0 |
| INT 2 | 0 |
| INT 3 | 0 |
| INT 4 | 0 |
| INT 5 | 0 |

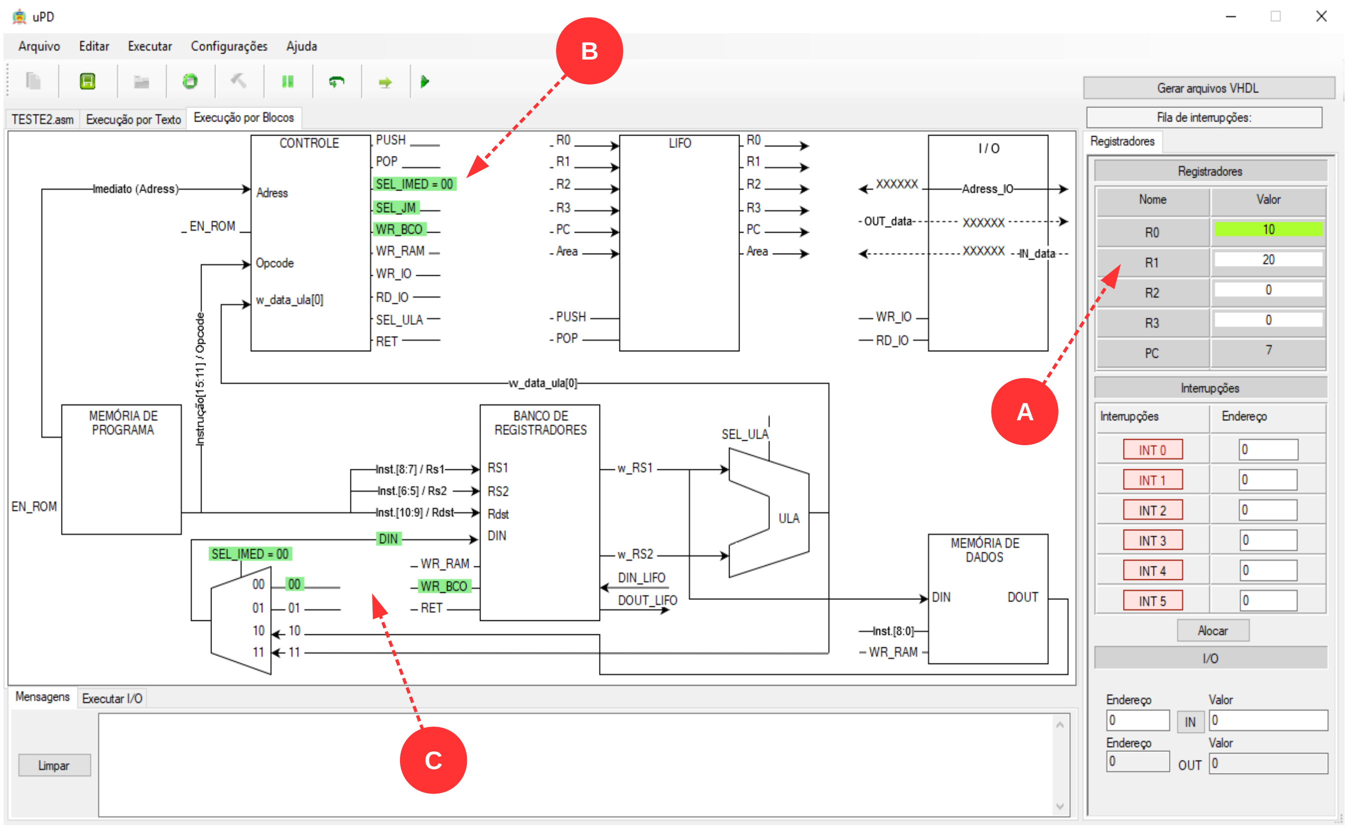
I/O

| Endereço | | Valor |
|---|---|---|
| 0 | IN | 0 |
| 0 | OUT | 0 |

*Fig. 5: Example of a text simulation.*

Figure 6 shows the Block simulation execution, in which it executes the same program that was done with the Text simulator. The letter A inside the red circle is pointing to the instruction LDI R1 20. Note that some signals at the output of Control block (pointed by the red circles with the letter B and C) were highlighted with colour green. It means that these signals will be active to ensure that this instruction flow through the data path.

All the simulations (Text and Block) are active as Windows Forms. It means that the user can check the execution step-by-step at Text simulator; meanwhile, he also checks the signal that was activated in the Block simulator.



| Nome | Valor |
|---|---|
| R0 | 10 |
| R1 | 20 |
| R2 | 0 |
| R3 | 0 |
| PC | 7 |

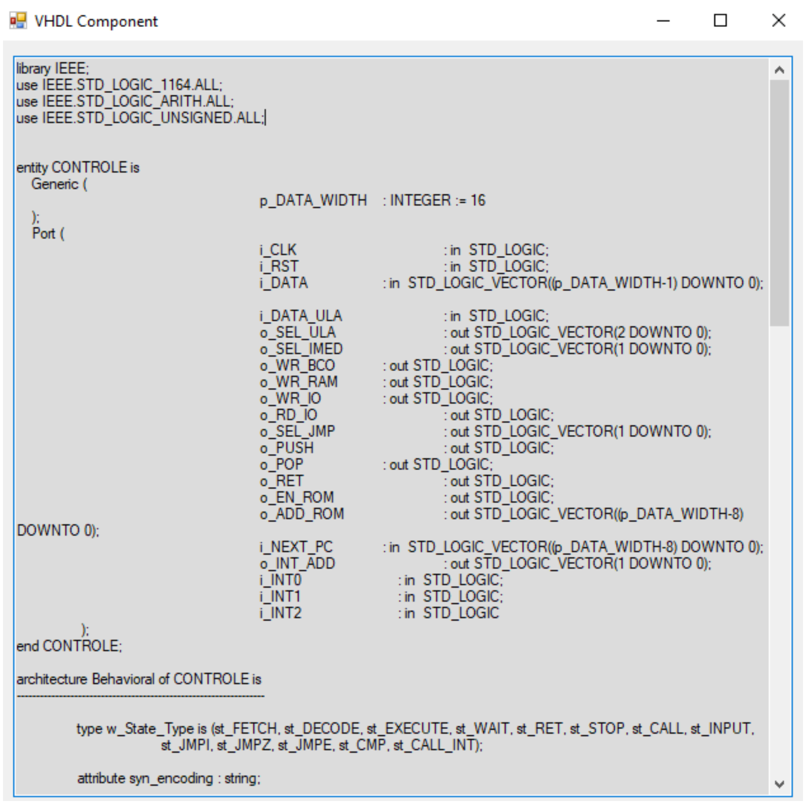
| Interrupções | Endereço |
|---|---|
| INT 0 | 0 |
| INT 1 | 0 |
| INT 2 | 0 |
| INT 3 | 0 |
| INT 4 | 0 |
| INT 5 | 0 |

*Fig. 6: Example of a block simulation.*

## VHDL code generation

As mentioned earlier, µPD was designed to be adopted as an interdisciplinary educational toolset, covering three courses of Computing Engineering: Computer Organization and Architecture, Very Large-Scale Integration (VLSI) Circuit Design, and Embedded Systems Design. That integration is possible due to the VHDL code generator, available in the µPD toolset. All processor internal blocks can be generated as a component, as shown if Figure 7, in which the control block was generated. These components may be synthesized with an FPGA manufacturers tools, like Quartus II, from Intel, or Vivado, from Xilinx. The interconnection of these components is part of the learning of the VLSI Circuit Design.

The users may simulate the processor as mentioned in previous subsections but also is possible to simulate only one processor component. It is a useful feature for the learning of students because they can understand the behavioural of each component generated by the tool. The ROM component is filled with the machine code related to the assembly program developed by the students.



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity CONTROLE is
   Generic (
                                    p_DATA_WIDTH    : INTEGER := 16
   );
   Port (
                                    i_CLK                   : in  STD_LOGIC;
                                    i_RST                   : in  STD_LOGIC;
                                    i_DATA          : in  STD_LOGIC_VECTOR((p_DATA_WIDTH-1) DOWNTO 0);

                                    i_DATA_ULA              : in  STD_LOGIC;
                                    o_SEL_ULA               : out STD_LOGIC_VECTOR(2 DOWNTO 0);
                                    o_SEL_IMED              : out STD_LOGIC_VECTOR(1 DOWNTO 0);
                                    o_WR_BCO        : out STD_LOGIC;
                                    o_WR_RAM        : out STD_LOGIC;
                                    o_WR_IO         : out STD_LOGIC;
                                    o_RD_IO                 : out STD_LOGIC;
                                    o_SEL_JMP               : out STD_LOGIC_VECTOR(1 DOWNTO 0);
                                    o_PUSH                  : out STD_LOGIC;
                                    o_POP           : out STD_LOGIC;
                                    o_RET                   : out STD_LOGIC;
                                    o_EN_ROM                : out STD_LOGIC;
                                    o_ADD_ROM               : out STD_LOGIC_VECTOR((p_DATA_WIDTH-8)
DOWNTO 0);
                                    i_NEXT_PC       : in  STD_LOGIC_VECTOR((p_DATA_WIDTH-8) DOWNTO 0);
                                    o_INT_ADD               : out STD_LOGIC_VECTOR(1 DOWNTO 0);
                                    i_INT0            : in  STD_LOGIC;
                                    i_INT1            : in  STD_LOGIC;
                                    i_INT2            : in  STD_LOGIC
          );
end CONTROLE;

architecture Behavioral of CONTROLE is
-----------------------------------------------------------------


        type w_State_Type is (st_FETCH, st_DECODE, st_EXECUTE, st_WAIT, st_RET, st_STOP, st_CALL, st_INPUT,
                    st_JMPI, st_JMPZ, st_JMPE, st_CMP, st_CALL_INT);

        attribute syn_encoding : string;
```

*Fig. 7: Example of a VHDL block generate by the tool.*

## IV. PRELIMINARY EVALUATIONS

There are different ways to measure the impact of a pedagogical resource on the students, and they can be a quantitative or qualitative way. The quantitative approach requires more numerical data information. For example, the authors [11] introduced a Simulator as a supporting tool for teaching the standard topics covered by an undergraduate course in computer architecture. They collected the percentage of success across the number of attempts to pass the final examination, over seven years: eight terms without the simulator they proposed, and six terms using their simulator.

On the other hand, a qualitative approach usually is based on the feedback (student's feeling) on the effectiveness of the adopted methodology. One example of this approach is the research done by [12], in which the authors evaluated the using of a simulator for teaching Computer Organization and Architecture. They applied a questionnaire about the proposed simulator, in which there were four options for each question: Agree, Disagree, Strongly Disagree, and Neutral.

We decided for a qualitative approach applying five questions to check if the students understood a common concept for the three courses (Computer Organization and Architecture, Very Large-Scale Integration – VLSI - Circuit Design, and Embedded Systems Design): "how the data flow" inside a processor, or inside an FPGA design, or in embedded system design using soft-core.

So far, we did evaluations of the tools with students of three mentioned courses (Computer Organization and Architecture, VLSI Circuit Design, and Embedded Systems Design), in two college terms. At the end of the college terms (last classes week), we apply an inquiry form with some questions to the students of three courses. The questions were the following:

1. Did you understand the data flow inside the µPD processor?
2. Did you understand why the µPD processor has a LIFO memory?
3. Did you understand the difference between program memory and data memory and how the processor deal with them?
4. Build a VHDL block helps on the understanding of how the processor works?
5. Was it easy to build embedded hardware using the µPD as soft-processor?

Students of Computer Organization and Architecture answered questions 1, 2, and 3. Students of the other two courses answered all of them. Each course had 20 students, which means 120 students after two college terms. Figure 8 shows the results of the inquiry form. To be classified as "understood" student must answer the five questions positively. One of the five questions answered negatively is classified as "confused". Hence, two or more is classified as "do not understand".

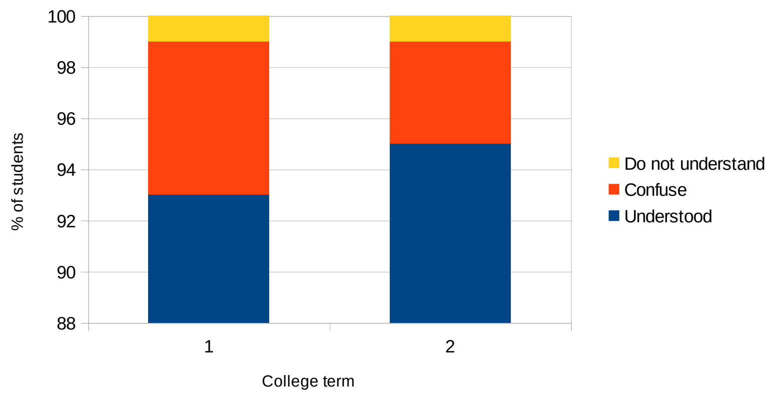


*Fig. 8: Students comprehension about µPD Processor.*

We also asked the students to list what they understand as positive in the µPD toolset, and they could state more than one. The features cited were: “Intuitive”, “Generate VHDL code”, “Easy to use”, “Simple and efficient for learning”, “Set of Instructions”, and “Few registers”. These features are depicted in Figure 9. Note that the majority of students considered the toolset as “intuitive”, and the second most positive feature was “Generate code VHDL”.

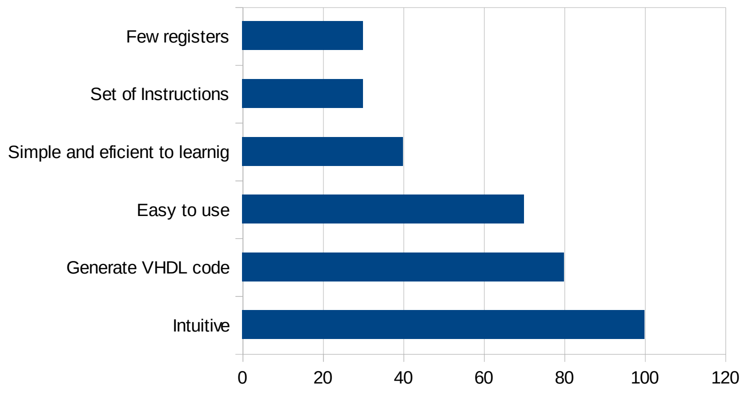


*Fig. 9: Positives features ranked by the students.*

## V. CONCLUSION AND FUTURE WORK

We understand that the toolset we have developed must be improved, on several points. For example, we think that the results related to students understanding depicted in Figure 8 should be 99%, at least. Is important to keep collecting opinions from the new students and use these opinions as feedback to improve the toolset.

Further the feedback, we will develop different components to be connected in the µPD external bus. These components were requirements asked by the students from the Embedded Systems Design course. According to them, more components will let more time to develop the embedded system that they must do up to the end of the term. They suggested components like a programmable SPI interface, a programmable I2C interface, and a programmable USART. Several sensors and actuators use one of these interfaces to be controlled by a host processor.

## REFERENCES

[1] P. Sanderson, K. Vollmar. MARS an Education-Oriented MIPS Assembly Language Simulator https//dl.acm.org/doi/10.1145/1124706.1121415, Accessed 13 March 2020.

[2] D. A. Patterson, J. L. Hennessy. Computer Organization and Design: The Hardware and Software Interface, Morgan Kaufmann – Elsevier, 2016.

[3] QtSPIM: a MIPS32 simulator. Available at: http://spimsimulator.sourceforge.net/ , accessed 15 March 2020.

[4] CPUlator: Computer System Simulator. Available at: https://cpulator.01xz.net/, Accessed 15 March 2020.

[5] R. N. Horspool, W. Lyons, and M. Serra. ARMSim# – a Customizable Simulator for Exploring the ARM Architecture, International Conference on Frontiers in Education: Computer Science and Computer Engineering, 2009.

[6] G. A. Esmeraldo, E. B. Lisboa. Uma Ferramenta para Exploração do Ensino de Organização e Arquitetura de Computadores. International Journal of Computer Architecture Education (IJCAE), v. 6, n. 1, p. 68-75, 2017.

[7] J. A. Borges, G. P. Silva. SimuS-Um Simulador Para o Ensino de Arquitetura de Computadores. International Journal of Computer Architecture Education (IJCAE) v, v. 5, n.1, p. 7-12, 2016.

[8] P. V. Vieira, A. L. A. Raabe, C. A. Zeferino. Projeto BIP: Impactos de 10 Anos de Uso de Uma Proposta Interdisciplinar de Ensino de Computação. International Journal of Computer Architecture Education, v. 5, p. 32-37, 2016.

[9] R. F. Weber. Fundamentos de arquitetura de computadores.Sagra Luzzatto , 2001.

[10] Z. Radivojevic, Z. Stanisavljevic, M. Punt, "Configurable simulator for computer architecture and organization", Comput Appl Eng Educ. 2018; 26: 1711– 1724. http://doi.org/10.1002/cae22034.

[11] D. Patti, A. Spadaccini, M. Palesi, F. Fazzino and V. Catania, "Supporting Undergraduate Computer Architecture Students Using a Visual MIPS64 CPU Simulator," in IEEE Transactions on Education, vol. 55, no. 3, pp. 406-411, Aug. 2012, doi: 10.1109/TE.2011.2180530.

[12] L. Null, and J. Lobur. 2003. MarieSim: The MARIE computer simulator. J. Educ. Resour. Comput. 3, 2 (June 2003), 1–es. DOI:https://doi.org/10.1145/982753.982754

[13] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, M. Sgroi. Benefits and challenges for Platform-based Design. Design Automation Conference – DAC, s. 1, s.n., 2004.

[14] D. Patti, A. Spadaccini, M. Palesi, F. Fazzino and V. Catania, "Supporting Undergraduate Computer Architecture Students Using a Visual MIPS64 CPU Simulator," in IEEE Transactions on Education, vol. 55, no. 3, pp. 406-411, Aug. 2012, doi: 10.1109/TE.2011.2180530.

[15] Prasad, P. W. C., Abeer, A., Azam, B., Chan, A.. "Using simulators for teaching computer organization and architecture.", Journal of Computer Applications in Engineering Education, V. 24, N. 2, 2016, doi:10.1002/cae.21699.

[16] J. Impagliazzo, R. Sloan, A. McGettrick, and P. Srimani, "Computer engineering computing curricula", in Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, Reno, Nevada, USA, February 19-23, 2003.

[17] J. Impagliazzo and A. N. Pears, "The CC2020 project — computing curricula guidelines for the 2020s," 2018 IEEE Global Engineering Education Conference (EDUCON), Tenerife, 2018, pp. 2021-2024, doi: 10.1109/EDUCON.2018.8363484.

[18] R. Hasan and S. Mahmood, "Survey and evaluation of simulators suitable for teaching for computer architecture and organization Supporting undergraduate students at Sir Syed University of Engineering & Technology," Proceedings of 2012 UKACC International Conference on Control, Cardiff, 2012, pp. 1043-1045, doi: 10.1109/CONTROL.2012.6334776.