Teaching Computer Architecture by Designing and Simulating Processors from Their Bits and Bytes

Mustafa Dogan[1,2], **Kasim Oztoprak**[3], **and Mehmet Resit Tolun**[4]

[1]**Aselsan Research, Ankara, Turkey**
[2]**Dept. of Computer Engineering, Hacettepe University, Ankara, Turkey**
[3]**Dept. of Computer Engineering, Konya Food and Agriculture University, Konya, Turkey**
[4]**Dept. of Software Engineering, Çankaya University, Ankara, Turkey**

Corresponding author:
Mustafa Dogan[1,2]

Email address: mustafadogan@aselsan.com.tr

ABSTRACT

Teaching Computer Architecture (Comp-Arch) courses in undergraduate curricula is becoming more of a challenge as most students prefer software-oriented courses. In some computer science/engineering departments, Comp-Arch courses are offered without the lab component due to resource constraints and differing pedagogical priorities. This paper demonstrates how students working in teams are motivated to study the Comp-Arch course and how instructors can increase student motivation and knowledge by taking advantage of hands-on practices. The teams are asked to design and implement a 16-bit MIPS-like processor with constraints as a specific instruction set, and limited data and instruction memory. Student projects include following three phases, namely, design, desktop simulator implementation, and verification using Hardware Description Language (HDL). In the design phase, teams develop their Comp-Arch to implement specified instructions. A range of designs resulted, eg, a) a processor with extensive user-defined instructions resulting in longer cycle times b) a processor with a minimal instruction set but with a faster clock cycle time. Next, teams developed a desktop simulator in any programming language to execute instructions on the architecture. Finally, students engage in Verilog Hardware Description Language (HDL) projects to simulate and verify the data-path designed during the initial phase. Student feedback and their current understanding of the project were collected through a questionnaire featuring varying Likert scale questions, some with a ten-point scale, and others with a five-point scale. Results of the survey show that the hands-on approach increases students′ motivation and knowledge in theComp-Arch course, which is centered around computer system design principles. This approach can also be effectively extended to related courses, such as Microprocessor Design, which delves into the intricacies of creating and implementing microprocessors or central processing units (CPUs) at the hardware level. Furthermore, the present study demonstrates that interactions, specifically through peer reviews and public presentations, between students in each phase increases their knowledge and perspective on designing custom processors.

1 INTRODUCTION

In an era of rapid technological advancement, understanding the intricate workings of computer hardware, from the basic building blocks to complex architectures, is essential. Recent advancements in computer technology have led to increased complexity and abstraction, making it imperative for individuals to have a comprehensive understanding of computer architecture (Hennessy and Patterson, 2017). This need is particularly pronounced within the context of the Comp-Arch course, an integral element of the Computer Engineering curriculum at Konya Food & Agriculture University (KFAU), and results in an intensified demand for individuals to possess an extensive comprehension of computer architecture, especially in educational contexts (Murdocca and Heuring, 1999).

Teaching Comp-Arch to undergraduate students presents a unique set of challenges (Theys and Troy, 2003; Kurniawan and Ichsan, 2017). In a world where many students gravitate towards software-oriented courses, the Comp-Arch course often appears as an insurmountable theoretical endeavor. Its demand for a profound comprehension of a computer's internal mechanisms can be daunting to students. The shift of students towards software-oriented courses has been observed in various academic institutions, posing a challenge for educators to instill a deep understanding of computer hardware (Ristov et al., 2011). However, understanding the Comp-Arch course's significance extends beyond academic achievement.

Bureau of Labor Statistics (hereafter called BLS) projects that jobs for Computer Hardware Engineers will grow by about 2% from 2020 to 2030, which is lower than the average growth (8%) (U.S. Bureau of Labor Statistics, 2022a). However, under the same conditions, BLS projects jobs for Information Security Analysts and Web Developers will grow by about 33% and 13%, respectively (U.S. Bureau of Labor Statistics, 2021, 2022b). Furthermore, BLS indicates that Computer Hardware Engineers get paid relatively more than Information Security Analysts and Web Developers (U.S. Bureau of Labor Statistics, 2021, 2022b,a). This paradigm shift in career preferences not only affects the Comp-Arch course but also extends to other core courses in the curriculum. Within the KFAU Computer Engineering program, the first three semesters include common core courses. Subsequently, during the following five semesters, students delve into 16 essential courses, including *Programming Languages*, *Computer Architecture*, *Operating Systems*, *Automata Theory and Formal Languages*, *Computer Networks* where students often suffer from the problem of balancing their growing interest in software-oriented careers with the demand for understanding computer hardware (Noone and Mooney, 2018; Thomas et al., 2012; Anderson and Nguyen, 2005; Vijayalaskhmi and Karibasappa, 2012; Prvan and OžEGOVIć, 2020). As a significant portion of the curriculum, these theory-driven and low-level computer comprehension courses are poised to shape students' career trajectories.

Despite these challenges, conventional pedagogical strategies have thus far failed to provide a practical and quantifiable approach to address these issues within the Comp-Arch course (Kurniawan and Ichsan, 2017; Theys and Troy, 2003; Nayak and Vijayalakshmi, 2013; Nova et al., 2013). This study presents a series of hands-on assignments meticulously crafted to invigorate students' enthusiasm for theoretical subjects and investigates the impact of these assignments on students' motivation and their ability to share knowledge among peers. These assignments include:

- Processor Design: Students are tasked with designing processors as depicted in Fig. 1, encompassing Instruction Set Architecture (ISA) and data-path components (Nayak et al., 2021).
- Desktop Simulator: Students create a desktop simulator that illustrates how processors execute assembly code and monitors memory and register content changes during execution (Brox et al., 2018).
- HDL Verification: The designed processors are verified using Hardware Description Language (HDL) (Velev, 2003).

Upon completing these assignments, students are encouraged to share their thoughts and insights through a survey, providing valuable feedback on the course's conduct. This approach not only enhances students' motivation and understanding of theoretical subjects but also encourages them to explore diverse scenarios and use cases within their coursework.

The significance of this 14-week study lies in its pioneering approach to bolster students' interest, motivation, and comprehension of the Comp-Arch course. First and foremost, this study presents a teaching model comprising the aforementioned assignments. This innovative teaching model offers a practical solution to a longstanding challenge and provides quantifiable results. Furthermore, it showcases the outcomes of student projects, with a multitude of teams each comprising 2-3 students. While a comprehensive presentation of all projects is beyond the scope of this study, we offer a compelling illustration of the remarkable progress achieved by showcasing two of these student projects. The study not only delves into the techniques employed by these students but also offers comprehensive documentation that is transparent and reproducible by fellow educators and researchers.

This innovative approach, which can be readily adapted to other courses, holds the potential to bridge the gap between theoretical knowledge and practical application. Moreover, it addresses the issue of low student engagement, especially in online or distance education settings (Sarder, 2014). Besides, if the requirements and constraints are arranged well, students can be more creative and focus on different aspects of user needs. By providing instructors with an effective assessment tool, it ensures that students receive the support and guidance needed to succeed in these challenging courses. Instructors can access and replicate this work shared on GitHub and YouTube, with small modifications to be used in their Comp-Arch classes.[1]

The remainder of this paper unfolds as follows: Section 2 provides the background, while Sections 3, 4, and 5 elucidate the design phase, simulator implementation phase, and hardware validation through Verilog. Section 6 delves into the results, analyzing data collected from student surveys. The ensuing discussion in Section 7 highlights the strengths and weaknesses of the project. Finally, the paper concludes in Section 8, with a discussion of future directions and potential extensions of this research.

2 RELATED WORK

Many scientific studies have been conducted on the difficulties encountered in Comp-Arch and other theoretical courses. One of the best ways to tackle these difficulties is to give students effective hands-on assignments, such as simulators (Wu et al., 2014). In this section, studies are analyzed in three main parts: a) courses that are hard to teach, i.e., computer architecture. b) possible problems and proposed solutions about why these courses are hard to teach. c) practicality and scope of simulators developed to teach Comp-Arch course.

Thomas et al., 2012 remark on difficulties and problems students can experience in a Comp-Arch course due to abstract concepts (Thomas et al., 2012). Simkins and Decker, 2016 and Omer et al., 2021 show that students grasp limited knowledge and the course itself becomes inefficient when teaching methods focus only on theoretical concepts (Simkins and Decker, 2016; Omer et al., 2021; Yehezkel et al., 2001; Kehagias, 2016). Anderson and Nguyen, 2005 survey the literature to find the best assignments for their course to avoid students struggling in the theoretical parts of an operating systems course (Anderson and Nguyen, 2005). Vijayalaskhmi and Karibasappa, 2012 state that teaching formal languages and automata theory courses is challenging due to the following reasons: a) monotonous teaching style b) courses mathematical nature causes poor understanding and students not showing adequate participation (Vijayalaskhmi and Karibasappa, 2012).

Leibovitch and Levin, 2011 mention difficulties faced in Comp-Arch courses due to the fact that Comp-Arch courses are comprised of different contiguous fields, such as digital design, embedded systems, operating systems, assembly programming, instruction set structure, and deciding, etc. (Leibovitch and Levin, 2011). On the other hand, Patel and Patt, 2019 state that the main issue is due to forcing students to memorize things before they understand the topic detailed (Patel and Patt, 2019). Simkins and Decker, 2016 survey the difficulties that students encounter during programming courses (Simkins and Decker, 2016). About 41% of students who encounter difficulties in "Tools for Learning" state that the main reason is lack of practice. Omer et al., 2021 collected and analyzed 66 different articles published from 2014 to 2020 to investigate recent developments in introductory programming course (Omer et al., 2021). Omer et al., 2021 and Malliarakis et al., 2016 suggest using games to increase students' motivation during the learning process (Malliarakis et al., 2016; Omer et al., 2021). Furthermore, hands-on experiences with processor architectures have a supportive impact on students' better understanding of the Comp-Arch course (Kehagias, 2016). In a comprehensive survey conducted by Kehagias, 2016, every assignment, ranging from fundamental conceptual questions to 130 complex programming assignments, was meticulously analyzed within the context of Comp-Arch courses offered at leading North American universities(Kehagias, 2016). The conducted research examines the quality and quantity of assignments to enlighten the pathway for educators and instructors to create assignments and thereby assess students properly. Kehagias, 2016 show that 25% of instructors include developing or modifying a simulator

[1] Playlist for MuSe and DoMe Architecture projects

design task for a target processor architecture, which is a core part of this approach (Kehagias, 2016).

The necessity of hands-on experiences in teaching different courses is examined in several studies and various assignments or projects are proposed to contribute to students' knowledge (Aviv et al., 2012; Hsu, 2015; Vijayalaskhmi and Karibasappa, 2012; Christopher et al., 1993). According to a survey by Omer et al., 2021 survey tools are needed to have sufficient visualization to help students comprehend subjects (Omer et al., 2021). In their work, Morgan et al., 2021 developed the RISC-156 V Online Tutor, which not only incorporates significant interaction signals within real hardware, sandboxes, knowledge checks, and challenges but also provides a RISC-V Integrated Development Environment (IDE) for assembly programming, as well as VHDL processor capture, simulation, and prototyping (Morgan et al., 2021). RISC-V IDE for assembly programming, and VHDL processor capture, simulation, and prototyping. Furthermore, the study by Wu et al., 2014 indicates that hands-on practices not only significantly improve students' scores in an introduction to computer science course but also lead to a reduction in students' stress levels during the course (Wu et al., 2014).

Nikolic et al., 2009 survey and evaluate the current simulators which are used to teach Comp-Arch course (Nikolic et al., 2009). The survey evaluates simulators in different categories which are the coverage of topics and features provided for simulation. The study shows that the best simulators that cover many topics are M5 and Simics simulators (Binkert et al., 2006; Magnusson et al., 2002). Several available simulators encompass approximately one-third of the Computer Architecture and Organization course content, focusing on fundamental aspects of Comp-Arch, memory system organization, architecture, interfacing, communication, device subsystems, and processor systems design. In contrast, M5 and Simics provide a more comprehensive coverage, spanning around two-thirds of the course material (Nikolic et al., 2009). Schuurman, 2013 developed a simulator to teach processor architecture basics to computer science students (Schuurman, 2013). Schuurman, 2013's approach shares common tasks with those advocated in this study, such as design and simulator phases. McGrew et al., 2019 delve into the creation of RISC-V processors on FPGAs, accompanied by a comprehensive toolchain for simulation, assembly programming, and C-level code compilation. They aim to empower undergraduate computer engineering students with practical experience in CPU design, emphasizing a solid grasp of processor internals and compiler operations, while highlighting the relevance of the RISC-V ISA in real-world applications (McGrew et al., 2019). Vollmar and Sanderson, 2005 introduce a Java-based MIPS assembly language simulator, specifically designed to cater to the needs of undergraduate computer science students and instructors, offering a user-friendly GUI and aligning with the straightforward design of the MIPS computer architecture commonly used in computer architecture and organization courses (Vollmar and Sanderson, 2005). Angelov and Lindenstruth, 2009 designed a 16-bit RISC-based non-pipelined processor which can be created by entry-level students as course homework (Angelov and Lindenstruth, 2009). Furthermore, they developed a simulator where users can type their assembly instructions and examine the code step by step. On the other hand, Bhagat and Bhandari, 2018 did not only design a 16-bit RISC processor but also, verified their design by using Verilog HDL (Bhagat and Bhandari, 2018). Similar to Bhagat and Bhandari, 2018, Angelov and Lindenstruth, 2009 also used Von Neumann architecture. However, their design is limited to support 15 different instructions to make the processor simpler and easier to design. Jaumain et al., 2007 distinguished themselves in the realm of microprocessor education by introducing a simulation platform. This platform offers students the capability to input assembly instructions and meticulously track the progression of each electric signal in a step-by-step fashion (Jaumain et al., 2007).

In their study, Rao et al., 2015 devised a processor design meticulously selecting components, including the ALU, control unit, and instruction decoder, with the goal of optimizing performance in terms of power efficiency and reduced delay. (Rao et al., 2015). However, they achieved these results using 32-bit instructions while the ALU can perform 16-bit operations.

Black, 2016 proposes a module to be used by students to allow them to run their designs in Arduino hardware with the help of an Emumaker86 simulator developed earlier by the professor (Black, 2016). The study concentrates on allowing students to run their code in hardware rather than designing a processor. Similarly, Yıldız et al., 2018 introduced VerySimpleCPU (VSCPU), a soft CPU simulation platform, alongside the offered tool, to enable students to not only design their processors from scratch but also construct code for their processors, and seamlessly implement them using FPGAs, collectively enhancing the educational experience in computer architecture and organization courses at the undergraduate level (Yıldız et al., 2018).

3 THE PROCESSOR (CPU) DESIGN

In this section, we first describe the foundational knowledge essential for understanding the MIPS architecture. Then, it defines the fundamental design limitations which each processor must support. Finally, it analyzes and compares the differences of each designed processor.

Before delving into the intricacies of the processor design, it's essential to gain insight into the students' educational background. The students who participated in this project possessed a strong foundation in mathematics, digital design, and basic Verilog knowledge, having completed prerequisite courses in Programming (Java, Python, C/C++), Discrete Mathematics, and Logic Design. While their programming skills were adequate for the development of the desktop simulator, the questionnaire responses revealed that they encountered challenges during the design phase involving HDL, primarily due to their limited experience with HDL programming.

In this project phase, students were tasked with defining key elements of their processor designs, including instruction architecture, data-path, control signals, supported instruction list, and the design of the arithmetic logic unit. Each group had a two-week period to brainstorm and conceptualize their architectural ideas. Following the principles highlighted by Omer et al., 2021 we fostered collaborative learning by having each group present their progress during various phases (Omer et al., 2021). During these presentations, students engaged in constructive discussions, providing feedback, and scrutinizing their peers' designs. This interactive process allowed for valuable insights and improvements in the project as they could leverage each other's strengths.

Initially, we had planned to introduce an additional phase focusing on the physical implementation of the students' designs. We believed this phase would significantly enhance their understanding of the subject matter. However, the outbreak of the COVID-19 pandemic forced the university and its facilities to close during the semester, resulting in the unfortunate cancellation of this phase of the project.

To differentiate between the two distinct CPU architectures created by separate student groups, we will employ the abbreviations MuSe and DoMe.

3.1 MIPS Architecture

This section provides a concise overview of the MIPS architecture. The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture, a renowned example of a RISC (Reduced Instruction Set Computer) design, offers a straightforward and efficient 32-bit word size with a load-store architecture (MIPS, 2001). It features 32 general-purpose registers and supports diverse instruction types for various operations (MIPS, 2001). This simplicity and regularity make MIPS an excellent choice for educational purposes, allowing students to grasp computer architecture. We chose MIPS for its advantages in enhancing students' understanding of computer architecture and organization.

When devising our CPU, we categorized instructions in a manner akin to the MIPS architecture, distinguishing between three primary types: R-type, I-type, and J-type instructions. Within the MIPS architecture, these instruction categories play a pivotal role, each with specific fields dedicated to fulfilling their unique functions (MIPS, 2001). The differentiation between these instruction types primarily hinges on their respective operation code (opcode) fields. Each instruction type possesses a distinct opcode value, except for R-type instructions. These R-type instructions, unlike their counterparts, do not encompass fields for target addresses, branch displacements, or immediate values. Instead, they feature fields for three registers, function codes, and shift amounts. The function code field plays a role in distinguishing R-type instructions from one another.

I-type instructions allocate the bits typically designated for the destination register, shift amount, and function code to represent immediate values. These immediate values serve various purposes, including acting as constant operands, branch target offsets, and memory operand displacements. I-type instructions offer users the convenience of incorporating constant values without necessitating a register. Meanwhile, J-type instructions facilitate alterations to program flow.

3.2 Prerequisites of CPUs

Since MIPS is a RISC type of architecture the students have limited instruction set, data and instruction memory, and register count. Each proposed architecture must support 18 predefined 16-bit instructions. These instructions are specified by the lecturer and given to students before the project starts. Students have separate 256-bytes program memory and data memory. Furthermore, they have eight 16-bit general purpose registers to access data that the CPU is currently processing. Each processor design must use a single cycle data-path and Von Neumann architecture to avoid complexity. With these specified instructions and constraints, architectures would be capable of writing many small-scale programs.

The CPU specifications were deliberately chosen to strike a balance between simplicity and comprehensiveness, leading us to opt for a 16-bit CPU design. While 8-bit CPU design was feasible, we excluded it due to its limited complexity, which might not align with the educational objectives of our course. On the other hand, the 32-bit CPU design was considered but ultimately dismissed because of the time constraints, as its development couldn't be completed within the designated time frame. These specifications were determined prior to the onset of the COVID-19 pandemic, with considerations for physical implementation that could potentially impact delivery schedules.

3.3 MuSe Architecture: The Processor Design Stage

This section provides a comprehensive overview of the MuSe Architecture, encompassing details on the Instruction Set, and Format. Furthermore, students delved into extensive work concerning ALU design, data-path development, and control signal configuration specific to their respective architectural projects. The MuSe Architecture is designed in such a way that it supports and conforms to all requirements mentioned in **Section 3.2 Prerequisites of CPUs**-at the same time, taking care of performance issues which are encountered in the DoMe Architecture. **Section A.1 MuSe Architecture Design** contain a more detailed exposition of these aspects.

3.3.1 MuSe Architecture: The Instruction Set and Format

Our proposed instruction format, detailed in Table 1, places an emphasis on optimizing support for mandatory instructions. Consistent with the MIPS architecture, we have organized instructions into three primary types: R, I, and J. When selecting fields for our instruction formats, we prioritized the utilization of the 3-bit opcode field as the ALU operation code, streamlining the instruction decoding process. This approach involves an initial assessment of the ***is_jump*** and ***is_imm*** fields to decode instructions efficiently. To minimize instruction decoding complexity, we sought to align 3-bit opcodes with corresponding ALU operation codes, thus reducing the time and hardware resources required to determine the specific ALU operation. We adopted an alternative approach, adding supplementary fields known as ***is_imm*** and ***is_jump*** each consisting of 1-bit length, to identify the instruction type. These added fields alleviate the need to rely solely on opcode distinctions, enhancing clarity in instruction classification.

R-type instructions involve the use of three register values, while I-type instructions employ immediate values, omitting the ***Rd***) field in favor of a 5-bit immediate field. J-format instructions, specialized for jumps, do not include the ***Rd***), ***Rs***, or ***Rt*** fields. Instead, they incorporate an 11-bit label field, serving as the storage location for target addresses. These distinctions align with their intended functions, rendering the ***is_jump*** field set to 1 for jump instructions and the ***is_imm*** field invariably set to 1 for immediate instructions.

For a comprehensive overview of the various instruction types supported by the MuSe Architecture, readers are encouraged to consult the **Section A.1 MuSe Architecture Design**.

3.4 DoMe Architecture: The Processor Design Stage

This section delves into the design principles of the DoMe Architecture and highlights its divergences from the MuSe Architecture. In contrast to the MuSe Architecture, DoMe Architecture places a strong emphasis on accommodating a broader range of instructions, extending beyond the mandatory set, including complex operations like division to enhance the CPU's functionality. While this approach does introduce some variance from the MIPS philosophy, it broadens the processor's capabilities, albeit with some associated performance trade-offs. Simultaneously, it affords students the opportunity to explore a wider spectrum of CPU functionalities and complexities.

3.4.1 DoMe Architecture: The Instruction Set and Format

In crafting the DoMe Architecture, designers drew inspiration from the traditional MIPS architecture, just as the MuSe Architecture did, thanks to MIPS's reputation for simplicity and user-friendliness (MIPS, 2001). Table 1 provides an overview of the DoMe Architecture's instruction format, which exhibits several distinctions from the MuSe Architecture in the following ways: 1) The DoMe Architecture employs a 5-bit function code. 2) It features an additional 1-bit ***control_bit***. 3) It omits the register address for the destination register ***Rd***. 4) Utilizes an 8-bit immediate value compared to MuSe Architecture's 5-bit value. 5) Dispenses with additional bits for ***is_imm*** and ***is_jump*** fields. 6) Forgoes the introduction of an extra instruction type for jump instructions.

In the R-type instruction format, the most significant 4 bits delineate the opcode, determining the instruction type. For R-type instructions, the CPU deciphers the instruction based on the function code. In other cases, the instruction is determined by the opcode, much like in the MIPS architecture. Following the 1-bit specification for the resulting register, the next 6 bits serve to specify the source and target registers, each represented in 3 bits due to the eight general-purpose registers available. The least significant five bits designate the function code for identifying R-type instructions.

In the I-type instruction format, the first three fields align with those in the R-type format, except for the inclusion of an immediate field in place of source register and function code fields.

While DoMe Architecture shares commonalities with the MIPS architecture, such as the presence of a function code, only two bits are allocated to the function code field, which may not sufficiently accommodate an extensive array of instructions under a single opcode. To promote diversity in projects, students' approach is supported by the instructor, allowing students to propose reducing one of the register addresses in the instruction format to free up more bits in the function code field. DoMe Architecture adopts a fixed, resulting register from among the eight general-purpose registers, instead of enabling the user to specify the register. Consequently, if a user seeks to perform an addition operation with the values of registers *r1* and *r2* and store the result in register *r3*, they must initially conduct the addition operation using registers *r1* and *r2*. As a result, the CPU stores the outcome in a default register called (***Rd***). Subsequently, the user must transfer the resulting value from (***Rd***) to the intended register. Additionally, DoMe Architecture introduces an alternative approach for result storage, granting users the ability to store the result in the target register using the ***control_bit*** specified in Fig. 2, allowing users to choose between these storage methods. If a user wishes to store the result in the destination register, the ***control_bit*** must be set to 1; otherwise, it should be set to 0. DoMe Architecture designers append the "***-c***" suffix to their instruction set to differentiate where the result is stored. This distinction is exclusive to R-type instructions and does not apply to I-type instructions, as I-type instructions only employ a single register.

The absence of a J-type instruction set is justified by the limited instruction memory of 256 Bytes in DoMe Architecture. The I-type format of the DoMe Architecture incorporates an 8-bit immediate part, facilitating access to any instruction within the instruction memory and enabling users to perform calculations with constants larger than 32 more conveniently. Further details on instructions in the DoMe Architecture are available in the **Section A.2 DoMe Architecture Design**.

4 THE SIMULATOR

Comp-Arch represents a multifaceted course that draws upon diverse elements from various domains of computer science, including operating systems and programming languages(Leibovitch and Levin, 2011). To optimize students' learning experiences, this course typically incorporates practical lab sections in which students can apply theoretical knowledge (Nikolic et al., 2009). Many activities within the Comp-Arch lab can be effectively executed through simulators, making simulators an indispensable pedagogical tool for teaching Comp-Arch (Burch, 2002; Djordjevic et al., 2005; Vollmar and Sanderson, 2006). As such, students are tasked with creating and presenting their simulators at different stages of the project, each of which must encompass the following essential features: 1) An interactive user input section. 2) The capacity to parse provided assembly code into machine code consistent with the processor's design. 3) The ability to visualize the current values of registers and memory cells. 4) Options for code interpretation, including step-by-step execution and full automation. 5) Freedom to employ programming languages and tools of their choice.

Araujo et al., 2014 have suggested a visualization approach, MIPS X-Ray, as an alternative to full-system simulation. However, students are expected to create simulators resembling the exemplary simulator diagram depicted in Fig. 3. The user interface plays a pivotal role in controlling and visualizing the simulator, comprising five key units: 1) The Instruction Decoder is tasked with receiving and decoding assembly instructions that adhere to the students' architecture through the user interface, with the decoded output seamlessly relayed to the Control Manager. 2) The Simulation Manager adeptly orchestrates the ALU and the Register and memory units, efficiently facilitating communication between these integral components. 3) The ALU, in turn, fulfills the crucial role of executing the requisite arithmetic calculations. 4) The Register and Memory unit plays its pivotal role by proficiently storing the contents of registers and memory, subsequently furnishing this vital data to the user interface. 5) The View Manager bears the responsibility of ensuring the timely and accurate updating of information stored within memory and register cells.

Preceding the design phase, students were tasked with constructing a MIPS simulator compatible with 32-bit MIPS instructions, encompassing all features of the original MIPS processor (MIPS, 2001). This initial simulator served a dual purpose: firstly, to deepen students' understanding of the MIPS architecture and provide a programming foundation for future simulators; and secondly, to grant students more time during the design and verification phases for developing desktop simulators tailored to their respective architectures. The choice of MIPS as the introductory architecture was underpinned by its innate simplicity, enhancing students' ability to design processors from scratch. Students were allocated three weeks to complete their individual simulators.

Subsequent sections delve into the distinctions between various simulators and scrutinize the impact of public presentations on simulator design.

4.1 MuSe Architecture: The Simulator

In the development of the MuSe Architecture simulator, the designers opted for the utilization of the Java programming language in conjunction with JavaFX for crafting an elegant graphical user interface. The simulator features a visually informative display of data memory, neatly presenting both the binary address and the associated value within that address in a user-friendly manner. To enhance comprehension, a dedicated section akin to the data memory is included for instruction memory, aiding in the visualization of machine code. Within the realm of the simulator's register visualization, the current register names and their respective values are thoughtfully portrayed in a signed decimal format. An additional segment serves to display the status of control signals during the execution of instructions, with highlighted cues signaling their activation. Furthermore, an explicit representation of the current program counter is complemented by an LCD Display that conveniently conveys crucial details encompassing the opcode, jump conditions, and immediate values of the ongoing instruction (Fig. 4). This inclusion of an LCD Display within the simulator was inspired by its real-world counterpart used in physical implementations. Since the simulator already offers visualizations of registers, the students thought it prudent to incorporate the LCD Display section as a software tool, facilitating a detailed exposition of the instruction currently undergoing execution. Notably, both students and educators can readily employ this simulator as an insightful tool for the analysis and exploration of architectural design, allowing them to make modifications in accordance with their requirements.[2]

For an in-depth exploration of MuSe Architecture's simulator details, we invite readers to delve into the intricacies presented in **Section B.1 MuSe Architecture Simulator**.

4.2 DoMe Architecture: The Simulator

In the realm of simulator development, a distinctive approach was undertaken by the creators of the DoMe Simulator, where the primary emphasis was placed on the comprehensive representation of data residing within memory cells and registers, as vividly illustrated in Fig. 5. Diverging from the language choice of MuSe, the DoMe Simulator was crafted using the Python programming language for the architectural framework and leveraged PyQt5 for the graphical user interface. A notable feature that sets the DoMe Simulator apart is its dynamic visualization of both memory and stack functionalities, enabling users to closely inspect and monitor alterations in these two critical components simultaneously. The presentation of memory cell contents within the DoMe Simulator distinguishes itself by offering not only binary representations but also decimal formats, which significantly eases the process of data analysis. Additionally, the DoMe Architecture designers took the extra step of offering multiple data representation formats within the registers, encompassing hexadecimal, signed integer, and unsigned integer formats, catering to diverse user preferences. Nonetheless, it's worth noting that the DoMe Simulator deviates from the MuSe Simulator in terms of visualizing control signals and lacks the LCD display section discussed in the previous section. Both students and educators can harness the capabilities of this simulator not only for utilization and replication but also for actively contributing to this open-source endeavor by introducing novel functionalities.[3]

To gain a profound understanding of the intricacies behind DoMe Architecture's simulator details, we encourage readers to immerse themselves in the comprehensive insights provided in **Section B.2 DoMe Architecture Simulator**.

5 VERIFYING THE CPU DESIGN USING A HARDWARE DESCRIPTION LANGUAGE

In the realm of Comp-Arch education, students embarked on a journey of enlightenment, immersing themselves in the intricate concepts of Comp-Arch and the art of instruction handling. Their academic voyage culminated in the meticulous design of a processor that encompassed vital elements, including the CPU, ALU, instruction formats, and datapath, thoughtfully crafted to adhere to a designated Instruction Set Architecture (ISA).

To illustrate their designs, students used a simulator to execute complex code sequences. The next step involved creating and simulating a Hardware Description Language (HDL) project that embodied their architectural concepts. While students had some flexibility, they leveraged conveniences provided by their Integrated Development Environment (IDE), such as 'for' loops and basic arithmetic operations.

This phase lasted four weeks to accommodate the students' foundational HDL knowledge. Most students favored using the Verilog hardware description language, and they implemented their HDL code in the Xilinx ISE Design Suite (Xilinx, 2007). Although Xilinx introduced the Vivado environment, students preferred ISE due to its lower system requirements.

Both student groups adopted the multiplication method proposed by Patterson and Hennessy, 2016, which handled negative values by converting them to positive and deciding the sign bit later. While the MuSe Architecture designers implemented this method in Verilog, the DoMe Architecture designers chose to use the "*" operand to minimize simulation risks.

In the forthcoming sections, we will delineate the modules employed by each architecture.

[2]MuSe Architecture Simulator
[3]DoMe Architecture Simulator

5.1 MuSe Architecture: The Verilog Design

The Verilog design of the MuSe Architecture, depicted in Fig. 6, is constructed using a set of seven distinct Verilog modules, each playing a crucial role in the overall functionality. These modules encompass: a) The Instruction Memory module is responsible for housing instructions provided by the user in binary format, the Instruction Memory module takes the program counter as its input and returns the forthcoming instruction slated for execution. b) The Data Memory module is a fundamental component that facilitates the management of data within the memory system. This module accepts data, addresses, and a suite of control signals as input and, in response, furnishes the data read from memory. c) The Register File module is a repository of registers, this module empowers users to effect changes in the register set. It acts as the gateway for user-initiated alterations to the register contents. d) The ALU module is the heart of the computation, the ALU module carries out a diverse array of operations, spanning addition, subtraction, multiplication, address calculations, and more. As outlined in the data-path structure, the ALU module receives two source inputs and a set of ALU control lines, using this information to execute the requisite computations and return the output value. e) The Control Unit module is the pivotal module that takes as input the opcode, ***is_jump***, and ***is_imm*** values, using them to dynamically update all associated control signals in a coordinated manner. f) The Processor Testbench module serves as the central orchestrator, assuming the role of the primary module, orchestrating the functionality of the other modules, and ensuring seamless and harmonious operation. g) The LCD Module is an interface module thoughtfully designed to aid users in the inspection of register and memory cell contents. The LCD module enhances the user's ability to visualize and comprehend the system's inner workings.

The Verilog code for the MuSe Architecture is available on GitHub.[4]

5.2 DoMe Architecture: The Verilog Design

DoMe Architecture's Verilog design shown in Fig. 7 utilizes Verilog modules. The followings are modules that DoMe Architecture designers created for their HDL project: a) Instruction Memory module contains instructions that are fed by a user in binary format. The module takes the program count as input and returns the instruction which is going to be executed. b) The GPRs module describes eight general-purpose registers and stores the values. This module helps us to read and write in registers. c) The Data Memory module contains an array with a length of 256 where each element represents 1-byte of data just like in the simulator. As in the GPRs module, the Data Memory module provides us with reaching and changing the content of given memory cells. d) ALU is a module where all the operations are done, such as summation, subtraction, multiplication, etc. As described in the data path, the ALU module takes two source inputs and one ALU control line array. According to these inputs, the ALU module carries out the necessary calculations and returns an output value. e) The Control Unit updates the control signals and ALU operation codes for every instruction according to their opcode and function code. The updated control signals are used in other modules like Data Memory, GPRs, etc. h) The RISC 16-Bit module is a container module to run the control unit module and data-path unit module together. These modules work simultaneously under the control of the RISC 16-Bit module.

The Verilog code for the DoMe Architecture is available on GitHub.[5]

6 RESULTS

In order to quantitatively assess students' experiences and perceptions throughout the project, a questionnaire was administered, utilizing a ten-point Likert scale (Likert, 1932). The questionnaire consisted of a set of questions, the average, and standard deviation results are presented in Table 2. To mitigate order effects, the questions were presented in random order, and students were allotted a sufficient amount of time (one hour) to complete the questionnaire to avoid procedural bias. The questions were meticulously crafted to ensure clarity and neutrality, preventing any leading question bias.

[4]MuSe Architecture Verilog
[5]DoMe Architecture Verilog

The table displays the mean responses and standard deviations for each question, shedding light on participants' experiences and perceptions both before and after the course. While the Likert scale questions (Questions 1-4) are rated on a scale from 0 to 10, the Likert-type questions (Questions 5-14) employ a five-point scale. It is worth noting that the Likert-type questions feature varied answer options; for instance, Questions 5-9 span from "Significantly Hindered (1)" to "Significantly Improved (5)". A more detailed analysis of all questions and answers is available in the **Section C Survey**. This table offers a comprehensive examination of the course's impact on participants' knowledge, skills, interests, satisfaction, and their inclination to pursue future opportunities in the fields of computer architecture and low-level systems.

The survey results reveal substantial improvements in students' knowledge, skills, and interests following the Computer Architecture course. Notably, students' self-reported knowledge and experience in hardware design demonstrated a significant increase, with the average score rising from 3.12 before the course to 7.02 after the course. Self-learning ability also saw remarkable enhancement, improving from an average of 4.64 to 7.52. Interest in low-level systems and Computer Hardware Engineering displayed a more moderate yet positive increase from 3.92 to 4.92. The course had a notable impact on participants' awareness of software development with consideration of hardware abilities and limitations. However, some questions received more mixed responses, such as the effect of group work, hands-on experiences, and the use of simulators and internet tools. These results suggest that the course was highly effective in increasing students' hardware design knowledge and self-learning ability, while also fostering an increased interest in low-level systems. While participants' overall satisfaction, confidence in understanding hardware design, and preparation for low-level system work were slightly improved, they generally expressed a high level of enjoyment with the course.

The interest in low-level systems and Computer Hardware Engineering showed a slight improvement rather than a significant one, which can be attributed to the variability in participants' responses, as indicated by the relatively higher standard deviation for these questions. Although the average ratings increased from 3.92 to 4.92, the larger standard deviations (2.50 to 2.89) signify greater divergence in students' responses. While some students experienced a substantial boost in their interest, others may have had more marginal changes or even a decline, resulting in an overall moderate increase in the average scores. This variance in individual responses could be due to various factors, including participants' prior interests, personal goals, and experiences, which influenced their perceptions of the course's impact on their interest in low-level systems and Computer Hardware Engineering.

In addition to the questionnaire analysis, a one-hour meeting was conducted among students and instructors to delve into the results and gather further insights. Several noteworthy points emerged from this discussion: a) Sequencing the implementation of the MIPS simulator before embarking on the CPU design phase significantly boosted students' confidence and awareness during the latter stage. b) Given that the project combines both programming and electronic skills, students exhibited a preference for collaborative group work over individual efforts. c) Participants observed that engaging in discussions and presentations at each phase of the project enhanced their understanding of the topics, even though this approach entailed additional workload. d) Due to COVID-19 constraints, students were unable to complete the physical implementation component, but they expressed a belief that conducting designs on an FPGA board instead of using ICs and breadboards would enhance their grasp of embedded systems. Additionally, with university laboratories equipped with ample FPGA boards, the physical implementation phase would be cost-effective compared to TTL implementation. e) Students attributed the enhancement of their design to their supervisor's meticulous guidance and consistent advice provided during presentations and discussions, resulting in a more foolproof project.

7 DISCUSSION

The Comp-Arch course offers a comprehensive curriculum, which can be challenging to cover in a mere 14-week duration. The three-phase processor design project successfully addresses the fundamental components of the course while providing students with a tangible experience. Through the use of discussions, peer reviews, presentations, and interviews during the project, abstract concepts are transformed into concrete knowledge. The course's flexibility allows instructors to tailor the scope to individual students' backgrounds and capabilities, potentially enhancing their learning experience. Nevertheless, to make the most of this course, students should ideally possess intermediate-level skills in programming, logic design, and HDL development.

The results of the post-project questionnaire align with previous observations and related research. Students emphasize the significance of the simulator (average rating of 4.04 out of 5) and the processor design phases (average rating of 3.78 out of 5). Simulators contribute to students' theoretical knowledge, while the processor design phase compels them to explore the functionalities of core components in the Von Neumann architecture, including the control unit and the logic unit. While the HDL implementation adds a fresh perspective to students' learning, it is important to acknowledge the substantial effort involved in this approach. Students also express concerns about the workload associated with this method.

Prior research often focused on either designing high-performance CPUs or creating CPUs and tools for educational purposes. Many studies revolved around demonstrating, implementing, and evaluating CPU designs from scratch. Additionally, various surveys targeted different courses, including Comp-Arch, to assess students' opinions and potential improvements to ongoing challenges. These surveys underscore the value of hands-on practical experience in theoretical courses. Instructors teaching the Comp-Arch course can readily adopt this approach, and students can access publicly available documents and videos. The approach's adaptability is one of its strengths and can be applied to various fields.

Despite our efforts to enhance the teaching of the Comp-Arch course, our study does not include a physical implementation phase. The physical implementation of designs in a lab setting offers valuable hands-on experience, enabling students to statistically evaluate their design's performance. Unfortunately, the COVID-19 pandemic prevented students from completing this phase, which can be considered for future enhancements. An FPGA implementation could be integrated as a final assignment to provide students with an opportunity to expand their practical workload.

Our study highlights that students engaged in hardware design and implementation often possess a foundational understanding of integrating software with hardware, taking into account the capabilities and limitations of the hardware, even before enrolling in the course. Simulating the assembly language instructions improves their comprehension of hardware and software interfaces, which will be invaluable in constructing intricate computational systems. Moreover, our results indicate that the three-phase processor design project has a slightly positive influence on students' career choices.

8 CONCLUSION & FUTURE WORK

The evolving landscape of computer science has seen students increasingly drawn to programming and related areas, despite the generally lower compensation in these fields compared to hardware engineering. Additionally, the complexity of certain courses poses challenges, as some students may struggle to grasp the material fully, potentially impacting their career prospects. As a response, this study delves into the effectiveness of a hands-on, team-oriented approach in motivating undergraduate students to engage with Comp-Arch, bolstering their knowledge in this domain. This approach particularly emphasizes the design and implementation of 16-bit processors across three project phases, evaluating its impact through a Likert scale-based questionnaire. In essence, our proposal introduces a three-phase processor design project for integration into Comp-Arch courses, with the aim of making it accessible to instructors and facilitating potential enhancements. This approach seamlessly blends creative thinking and hands-on practice while incorporating peer reviewing and public presentations. It can serve as a foundational model for other computer engineering courses that require a solid grasp of low-level computer understanding. This article offers insight into students' processor design experiences and highlights the knowledge diffusion that occurs within teams as they concentrate on different CPU aspects. The study underscores that, with careful guidance, students can successfully design fully functional processors and their corresponding assembly languages. The gain of the present study can be summarized as: *"Comp-Arch course would be very interesting* *and benefits when proper tools and assignments are provided.".*

In terms of future work, this study opens avenues for scientific exploration and educational enhancements. Researchers can explore the scope of this approach within the Comp-Arch course, suggesting alternate approaches to cover diverse content, such as pipelining, while maintaining student motivation. Addressing potential tool-related difficulties, as noted by Omer et al., 2021, researchers can introduce more user-friendly tools and assess their effects on students' development, as exemplified by RISC-V Online Tutor (Morgan et al., 2021). Furthermore, the study acknowledges the high workload placed on students by this approach. Future investigations could break down the effect of each project phase, quantifying the workload-to-benefit ratio. This insight can guide educators in achieving comparable results with reduced student workload. Finally, considering students' keen interest in FPGA board-based designs to enhance their understanding of embedded systems, a promising avenue for future work could involve integrating the further development and evaluation of this FPGA-based implementation approach as an additional phase within the existing three-phase project structure. Leveraging university resources with ample FPGA boards, this approach could prove to be cost-effective and align well with contemporary advancements in embedded systems education.

ACKNOWLEDGMENTS

The authors express their gratitude to Ömer Metin, Mustafa Çataltaş, and Sevcan Doğramacı for their valuable contributions to the design and reporting processes (Dogan, 2023; Metin and Dogan, 2023; Dogramaci and Cataltas, 2023a,b). We have provided access to all codes and YouTube videos through the following links:

- Playlist for MuSe and DoMe Architecture projects
- MuSe Architecture Simulator
- DoMe Architecture Simulator
- MuSe Architecture Verilog
- DoMe Architecture Verilog

SUPPLEMENTARY

The supplementary material is divided into three distinct sections: **Section A Architecture Details** offers comprehensive information on the MuSe and DoMe architectures, encompassing details on the multiplication algorithm, ALU design, and data path control signals. In **Section B Simulator Details** we provide a detailed breakdown of the simulator structures through class descriptions. Lastly, **Section C Survey** delves into the specifics of the survey questions and the corresponding responses.

A ARCHITECTURE DETAILS

A.1 MuSe Architecture Design

In this section, we present detailed information on the multiplication algorithm, ALU design, data path, and control signals within the MuSe Architecture. The architecture's instructions, as outlined in Table 1, are further elaborated in Table 3, providing a comprehensive explanation for each instruction. Notably, the MuSe Architecture includes an instruction referred to as ***syscall***, which, while not mandatory for simulator functionality, serves the purpose of enabling programmers to display register contents on an LCD screen when invoked. It's worth mentioning that the ***jr*** and ***syscall*** instructions are exceptions within the architecture, as they do not make use of target and destination registers; however, they are categorized under the R-Format due to their utilization of source registers.

A.1.1 MuSe Architecture: The Multiplication Algorithm

Multiplication operations can be computationally intensive, often requiring significant processing time within the Arithmetic Logic Unit (ALU). In the MuSe Architecture, our approach to handling multiplication tasks follows the iterative algorithm proposed by Patterson and Hennessy, 2016. This methodology implements multiplication iteratively, utilizing registers and adders, as depicted in Fig. 8. To accommodate this iterative approach, the MuSe Architecture incorporates a specialized multiplication unit within the ALU. This unit comprises eight shift registers and eight 8-bit adders, tailored to efficiently handle multiplication operations.

A.1.2 MuSe Architecture: The ALU Design

The Arithmetic Logic Unit (ALU) serves as the central component of the CPU, responsible for executing all computational tasks. Within the MuSe Architecture, the ALU manages its operations through two input ports, referred to as input A and input B, along with an output port named output C. The ALU is highly versatile and capable of performing a wide array of calculations. The type of calculation to be executed is determined by the 3-bit ALU control lines denoted as F0, F1, and F2. These control lines are governed by the ALUOp control signal, which is introduced in the data path and changes dynamically as each incoming instruction is decoded.

Table 4 presents a comprehensive list of mathematical operations supported by the MuSe Architecture, along with their corresponding control line values. MuSe Architecture's ALU design accommodates eight distinct operations, each represented by specific combinations of these three control lines.

A.1.3 MuSe Architecture: The Data-path and Control Signals

The data-path within a processor serves as the central framework that interconnects essential hardware components, including functional units like the ALU, adders, memory, and registers. It is also equipped with a set of control signals that facilitate the coordination of different CPU units. For instance, when the load word (*lw*) instruction is executed, setting the memory read control signal to 1 enables reading data from memory. When designing the data-path for MuSe Architecture, the developers drew inspiration from the original MIPS data-path.

In the conventional MIPS architecture, there are eight distinct control signals (MIPS, 2001). However, in the design of MuSe Architecture, 11 control signals are integrated to accommodate various instructions effectively. These additional control signals, in addition to those inherited from traditional MIPS architecture, include the System Call (***syscall***), Jump Register (***JumReg***), and Shift Register (***Shift Reg***) control signals. Control signal values for each instruction in MuSe Architecture are thoughtfully compiled in Table 5, aiming to facilitate and guide researchers in replicating the MuSe Architecture design.

A.2 DoMe Architecture Design

In this section, we delve into an in-depth exploration of the multiplication algorithm, ALU design, data path, and control signals incorporated within the DoMe Architecture.

The instructions utilized in the DoMe architecture are thoughtfully laid out in Table 3, accompanied by their usage in desktop simulators. Notably, it is important to highlight that the suffix "***-c***" is exclusive to the application in R-type instructions.

A.2.1 DoMe Architecture: The Multiplication Algorithm

In this section, we delve into the multiplication algorithm employed in the DoMe Architecture design phase. Following extensive research, the DoMe Architecture designers initially opted for the Wallace Tree multiplication method (Wallace, 1964). However, during the design phase presentation, their project co-advisor recommended an alternative multiplication approach. The advisor's advice was grounded in the realization that implementing an 8-bit Wallace Tree multiplier would entail significant costs and complexities, potentially posing challenges during the planned physical implementation phase. Consequently, the DoMe Architecture designers pivoted to adopt the multiplication algorithm suggested by the MuSe Architecture in **Section A.1.1 MuSe Architecture: The Multiplication Algorithm** to enhance feasibility and practicality.

A.2.2 DoMe Architecture: The ALU Design

DoMe Architecture has been meticulously crafted to provide support for a broader spectrum of instructions compared to the MuSe Architecture. Consequently, the repertoire of DoMe

Architecture instructions includes division and exclusive or (***xor***) operations, in addition to the other eight operations. Recognizing the challenge of representing ten different operations with merely three control lines, the DoMe Architecture designers astutely incorporated an additional control line (F3). This design choice resulted in six unused control signals. Further elaboration on ALU operations and their corresponding control line values can be found in the supplementary materials. Furthermore, the ALU operation list for the DoMe Architecture, as delineated in Table 4, encompasses a more extensive array of operations and includes the ALU control lines (F3), a notable expansion in comparison to MuSe Architecture's ALU Design.

A.2.3 DoMe Architecture: The Data-path & Control Signals

In this section, we delve into the DoMe Architecture's distinctive approach to designing its data-path and control signals. As previously highlighted in Section 3.2.4, the traditional MIPS architecture encompasses eight distinct control signals. However, the MuSe Architecture augmented this count to 11. In contrast, the DoMe Architecture streamlined its control signals to a total of seven, achieved by the elimination of the Register Destination (RegDst) and Memory to Register (MemtoReg) control signals. An additional control signal, known as "jump," was introduced to facilitate the management of jump instructions. To accommodate DoMe architecture, all unit and control signal connections in the MIPS data-path were meticulously revised. Much like in the MuSe Architecture, the control signal values pertinent to DoMe Architecture can be found in Table 6.

B SIMULATOR DETAILS

B.1 MuSe Architecture Simulator

The architecture of the MuSe simulator is intricately composed of several well-defined classes, each dedicated to a specific aspect of the simulation. These classes encompass ALU, Controller Unit, Instructions, Instruction Memory, Data Memory, Register File, Program Counter, and Processor. Within the ALU class, the designers have favored the use of elementary operators for computational tasks rather than embarking on the intricate endeavor of implementing complex logic circuits such as full adders and multiplication logic. The Controller Unit class assumes the pivotal responsibility of assigning control signals to individual instructions through a meticulous evaluation of opcodes, ***is_jump*** flags, and ***is_imm*** values. The MuSe simulator primarily revolves around three distinct instruction types, namely R-type, I-type, and J-type instructions. To streamline the management of shared attributes across these instruction types, the simulator extends these instructions from an abstract class named Instruction. The Instruction Memory is constructed as an array of instructions, initialized upon program commencement. Meanwhile, the Data Memory module houses a two-dimensional byte array augmented with a stack pointer, effectively governing all essential memory read and write operations. The Register File class boasts a roster of registers, encompassing eight pre-defined registers and proficiently manages both read and write operations in alignment with control signals. The Program Counter class is characterized by its simplicity, storing merely an integer value to represent the program counter value and adeptly governing its manipulation. The Processor class orchestrates the harmonious interaction of the previously mentioned classes, meticulously choreographing their workflow. The program commences with the initialization of instruction memory, and as it unfurls, the processor diligently fetches instructions in a loop. Notably, in the MuSe architecture, the absence of a pipeline implementation means that the simulator only progresses to the next instruction once the current instruction is successfully executed. The program culminates either when all instructions are executed or in the event of an encountered error.

B.2 DoMe Architecture Simulator

The intricate structure of the DoMe Simulator consists of an array of classes, with certain classes sharing functional similarities with their counterparts in the MuSe Simulator. The essential classes used in the DoMe Simulator include Registers, Instructions, Data Memory, Instruction Memory, Definitions, Instruction Functions, Assembler, and Processor. The Registers class, much like in the MuSe Simulator, serves as the repository for registers, with their initialization taking place here. The Instructions class stores the assembly instructions within an array and enables their retrieval by the Processor class through a systematic loop. Data Memory, much like its MuSe counterpart, plays a pivotal role in memory operations, with memory contents being stored within an array. It's noteworthy that, unlike the MuSe Simulator, the DoMe Simulator foregoes the inclusion of a dedicated ALU class. Instead, it features the Instruction Functions class, which defines a function for each instruction operation and establishes a linkage between function and instruction. This unique approach allows the simulator to directly access these functions and execute the requisite operations. The Assembler class within the DoMe Simulator shares similarities with the Control Unit class in the MuSe simulator, responsible for the assignment of control bits, opcodes, registers, and other essential values. The Definitions class serves as a valuable look-up table, pre-defining the properties of each instruction and is instrumental in aiding the Assembler class in the configuration of instruction attributes. Despite disparities in the implementation of specific components within their data-path, the designers of the DoMe Simulator are acutely aware that this aspect of the project places a distinct focus on augmenting the capacity and workflow of their architecture.

C SURVEY

This section delineates the survey questions and presents the responses provided by 50 students.

Question 1 aims to assess participants' perception of their hardware design knowledge and experience, both before and after the course (Fig. 9).

Question 2 measures the participants' self-reported self-learning ability before and after the course, which is a vital aspect of their educational growth (Fig. 10).

Question 3 gauges the change in participants' interest in working on low-level systems like Computer Hardware Engineering from before to after the course, helping understand how the course impacts their preferences (Fig. 11).

Question 4 investigates participants' awareness of developing software considering hardware limitations before the course, offering insights into their initial perspectives (Fig. 12).

Question 5 focuses on the impact of visual tools on the participants' learning experience throughout the course, examining how such tools influenced their engagement (Fig. 13).

Question 6 explores the influence of group work on the learning experience, uncovering the effect of collaborative learning on participants' perceptions (Fig. 14).

Question 7 assesses how hands-on experiences during the course influenced participants, providing insights into the practical aspects of their learning (Fig. 15).

Question 8 evaluates the effect of designing their own architecture on participants' learning experience, highlighting the significance of practical application in the course (Fig. 16).

Question 9 measures the impact of simulators and internet-shared tools on participants' learning experience, indicating the relevance of digital resources (Fig. 17).

Question 10 evaluates the overall satisfaction level of participants with the Computer Architecture course, summarizing their general contentment (Fig. 18).

Question 11 investigates the alignment between participants' expectations and the course's content and delivery, offering insights into how well the course met their initial anticipations (Fig. 19).

Question 12 assesses participants' confidence in their understanding of hardware design after completing the course, indicating the level of self-assuredness in their knowledge (Fig. 20).

Question 13 aims to measure how well participants believe the course prepared them for low-level system work like Computer Hardware Engineering (Fig. 21).

Question 14 captures participants' overall enjoyment of the course, providing a holistic perspective on their satisfaction and engagement with the educational experience (Fig. 22).

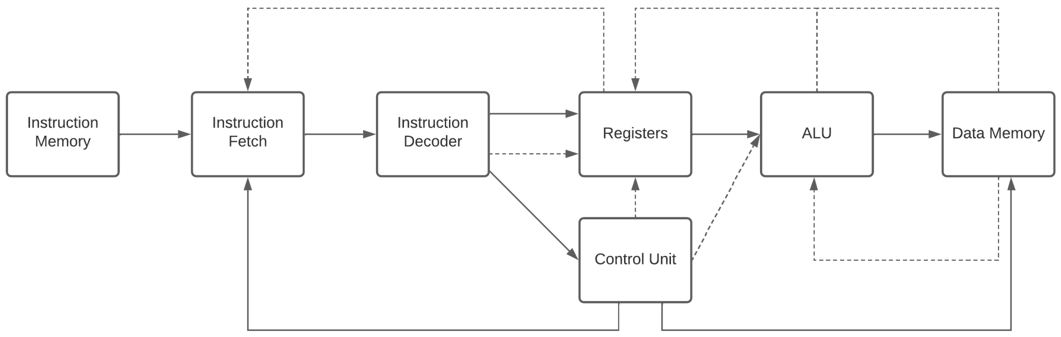
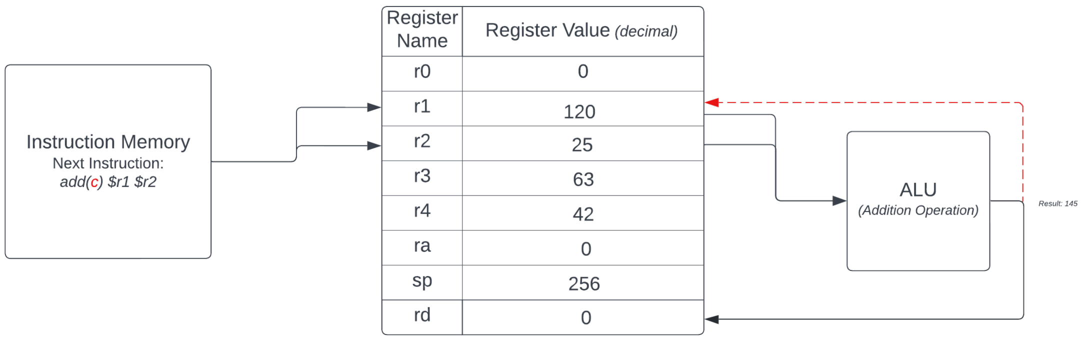


Figure 1. Block-Diagram of a Simple CPU.



Register Name	Register Value *(decimal)*
r0	0
r1	120
r2	25
r3	63
r4	42
ra	0
sp	256
rd	0

Figure 2. DoMe Architecture Control Bit Example

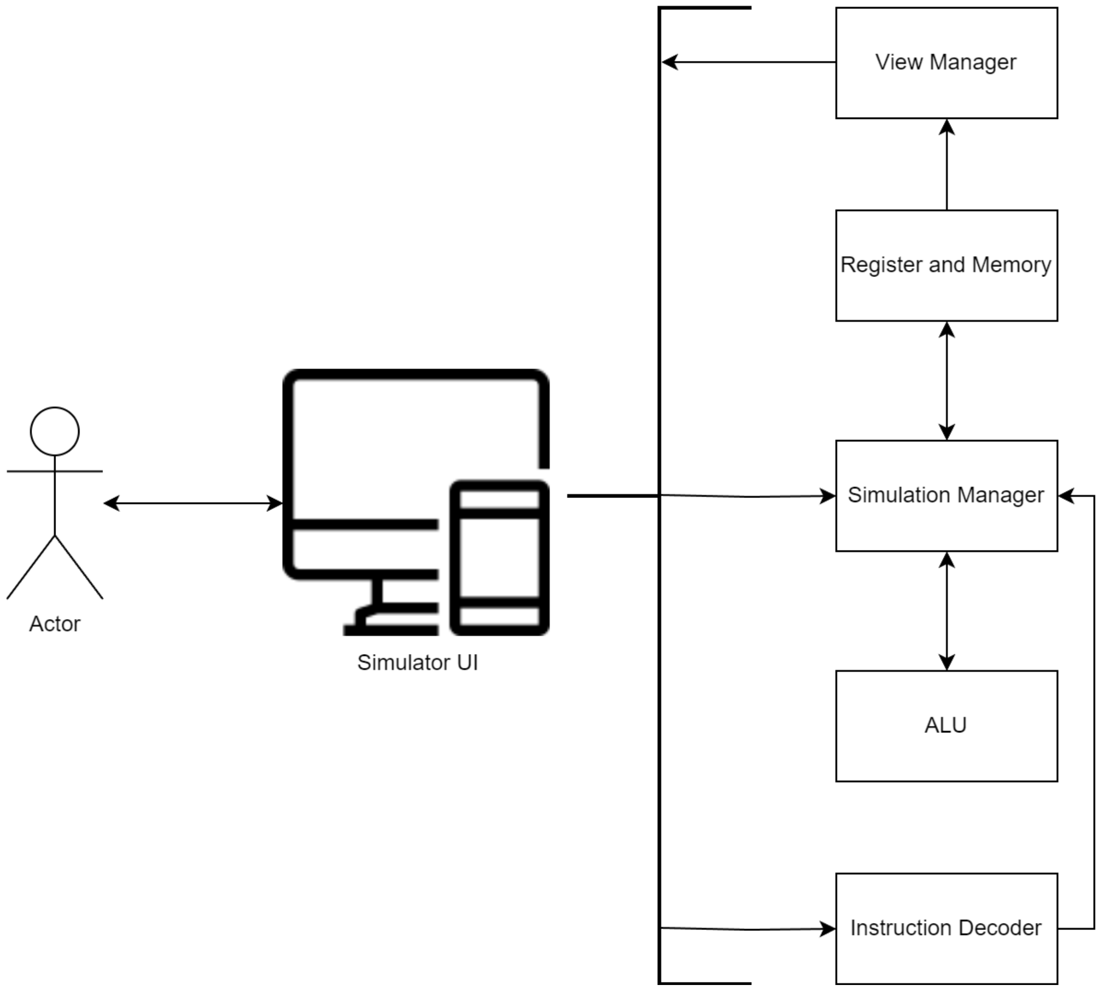
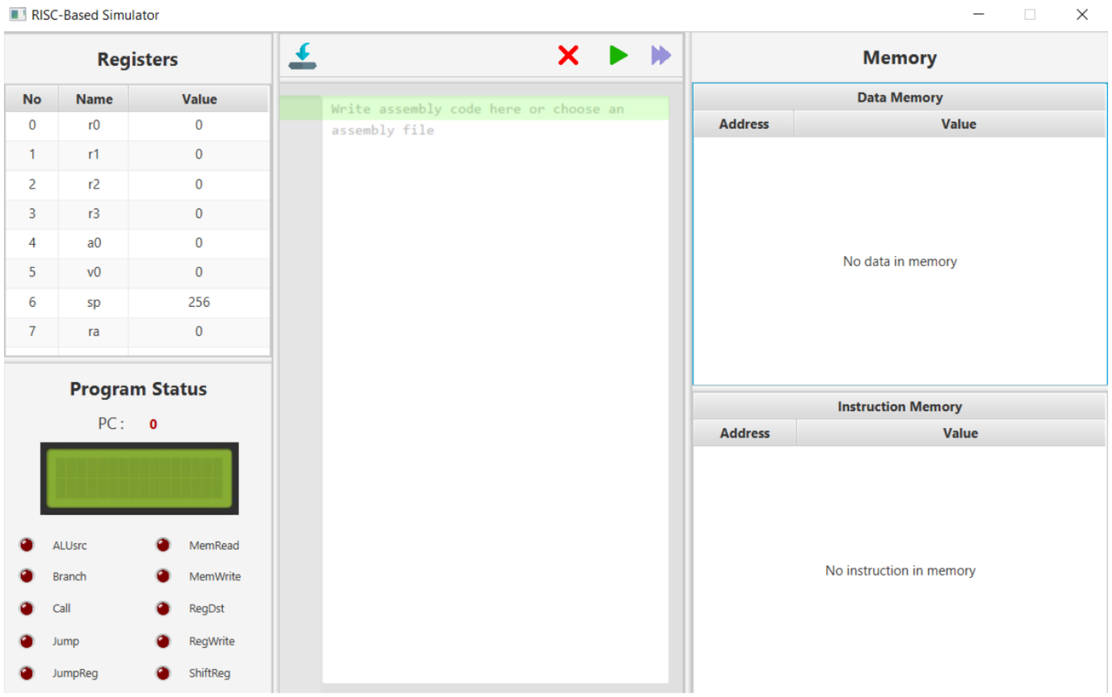


Figure 3. Example Simulator Diagram Used for Teaching Comp-Arch Course



No	Name	Value
0	r0	0
1	r1	0
2	r2	0
3	r3	0
4	a0	0
5	v0	0
6	sp	256
7	ra	0

PC : 0

ALUsrc	MemRead
Branch	MemWrite
Call	RegDst
Jump	RegWrite
JumpReg	ShiftReg

Data Memory

Address	Value

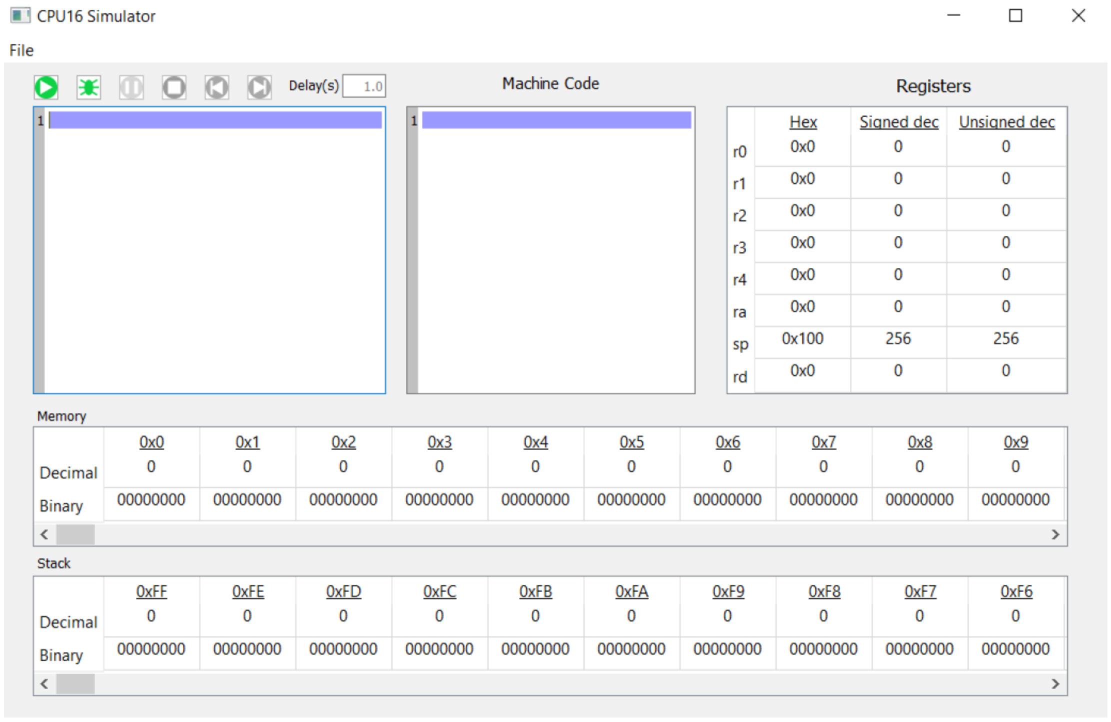
No data in memory

Instruction Memory

Address	Value

No instruction in memory

Figure 4. MuSe Architecture Desktop Simulator Screen.



	Hex	Signed dec	Unsigned dec
r0	0x0	0	0
r1	0x0	0	0
r2	0x0	0	0
r3	0x0	0	0
r4	0x0	0	0
ra	0x0	0	0
sp	0x100	256	256
rd	0x0	0	0

Memory

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9
Decimal	0	0	0	0	0	0	0	0	0	0
Binary	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Stack

	0xFF	0xFE	0xFD	0xFC	0xFB	0xFA	0xF9	0xF8	0xF7	0xF6
Decimal	0	0	0	0	0	0	0	0	0	0
Binary	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure 5. DoMe Architecture Desktop Simulator Screen.

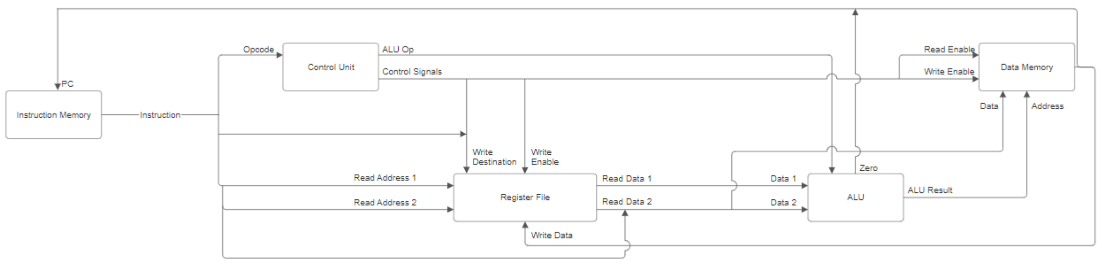


Figure 6. Input and Output of Verilog Modules in Muse Architecture.

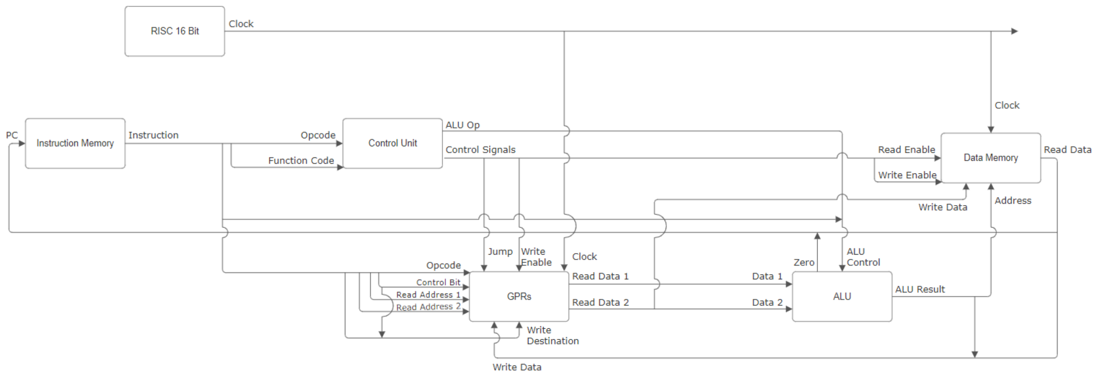


Figure 7. Input and Output of Verilog Modules in DoMe Architecture.

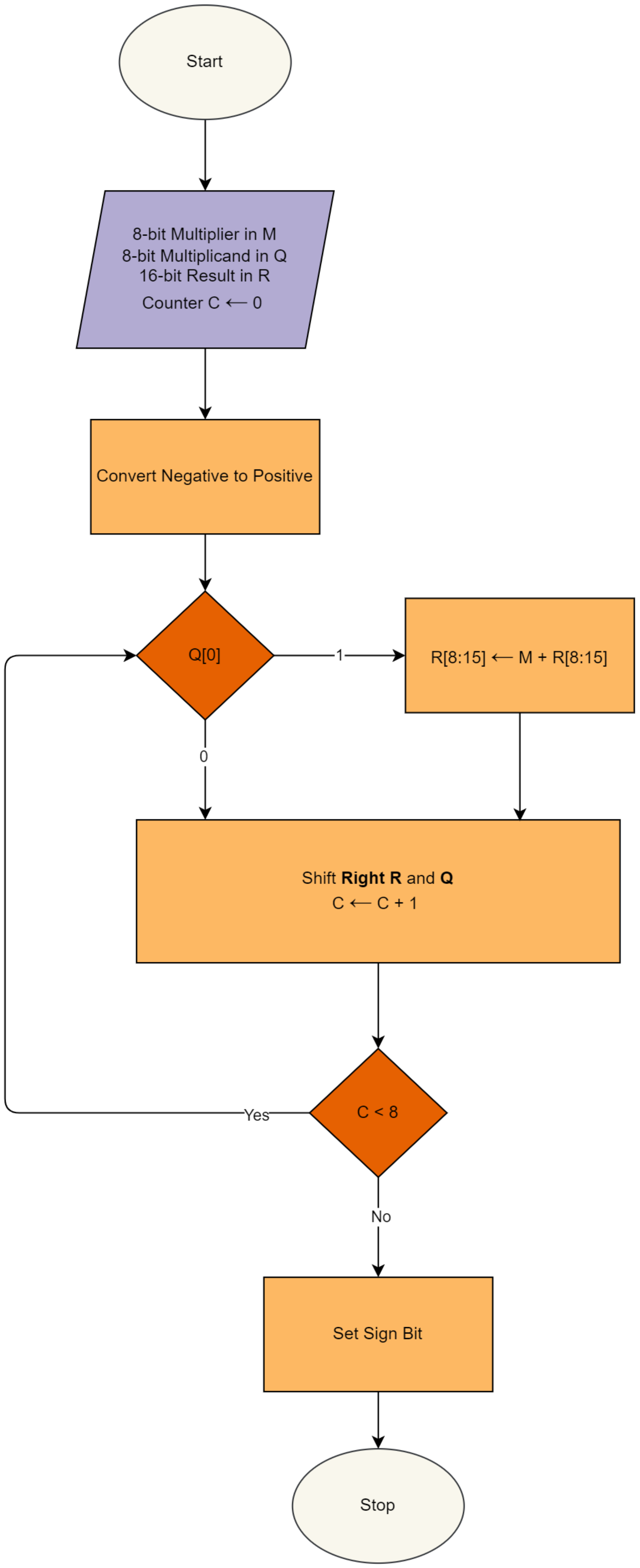


Figure 8. Multiplication Algorithm Flow Chart.

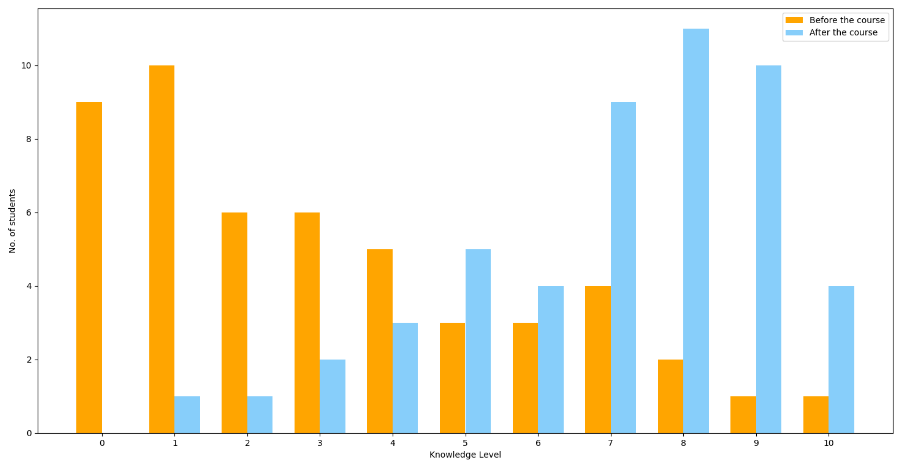


Figure 9. Results for *"How would you rate your knowledge and experience level about hardware design before/after the course?"*

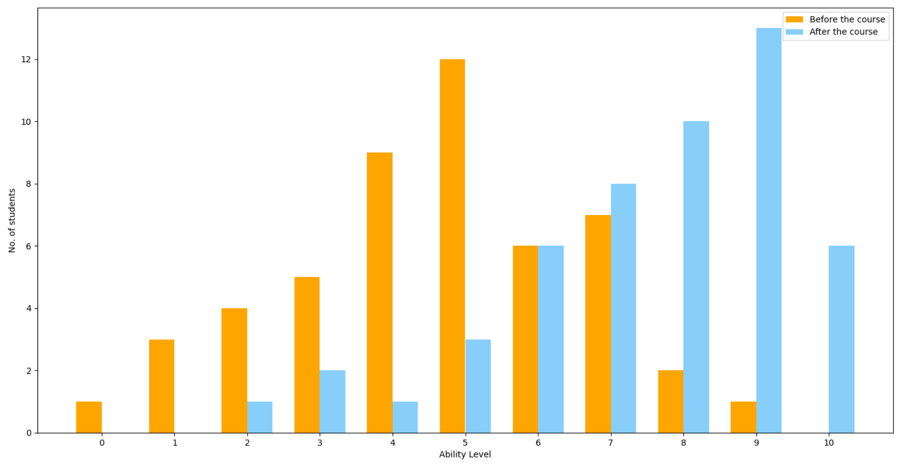


Figure 10. Results for *"How would you rate your self-learning ability before/after the course?"*

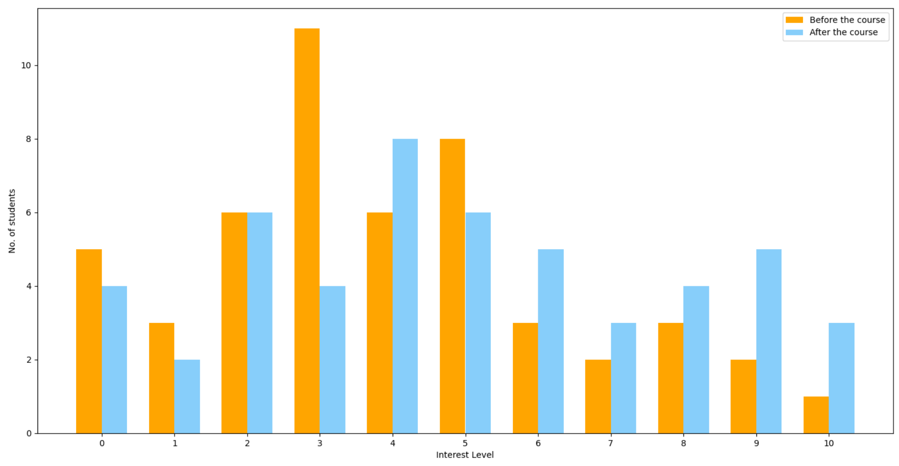


Figure 11. Results for *"How interested were you in working on low-level systems, such as Computer Hardware Engineering before/after the course?"*

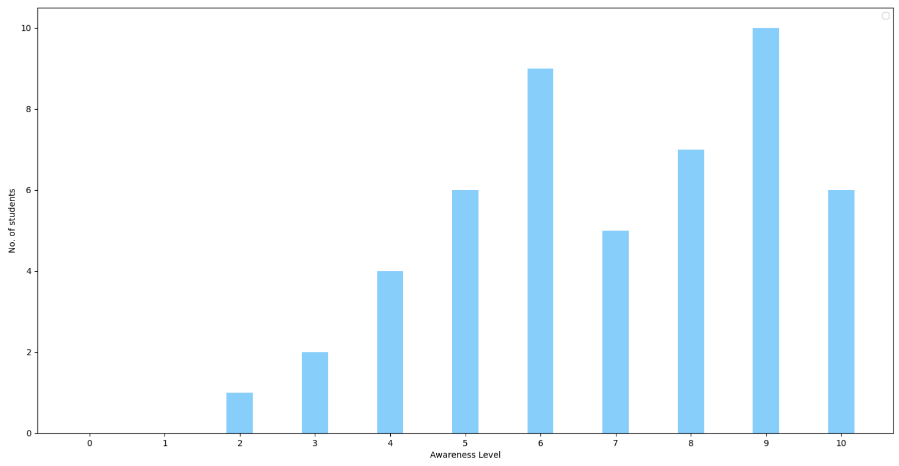


Figure 12. Results for *"How aware were you of developing software by considering hardware abilities and limitations before the course?"*

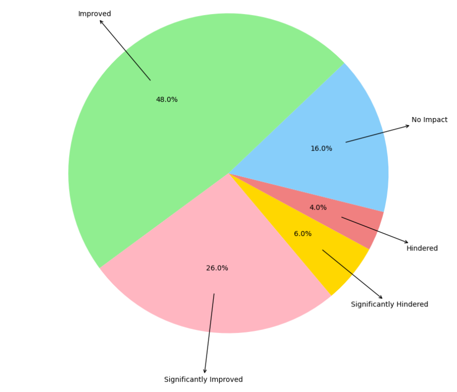


Figure 13. Results for *"How did the use of visual tools during the course impact your learning experience?"*

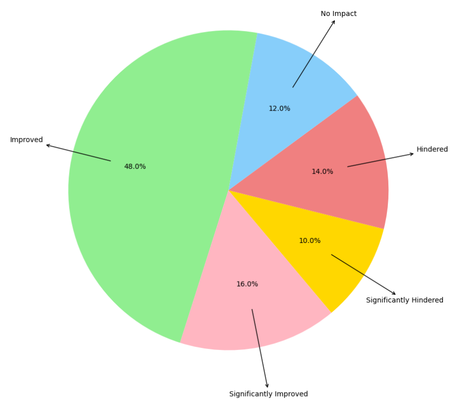


Figure 14. Results for *"How did working in groups during the course affect your learning experience?"*

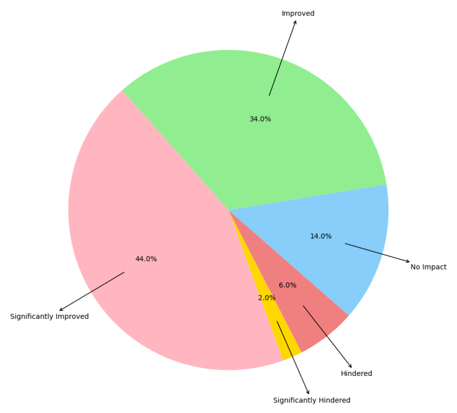


Figure 15. Results for *"How did hands-on experiences during the course influence your learning experience?"*

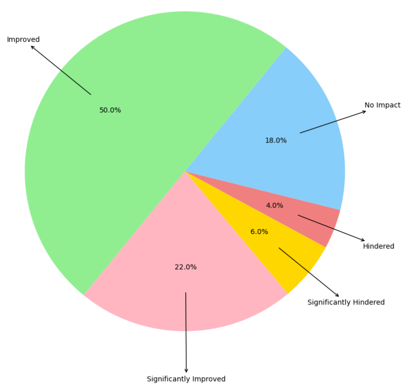


Figure 16. Results for *"How did implementing and designing your own architecture during the course affect your learning experience?"*

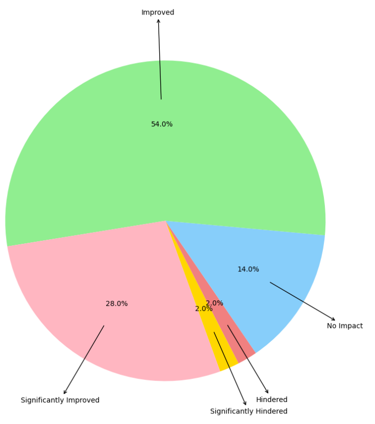


Figure 17. Results for *"How did using simulators and tools shared on the internet during the course impact your learning experience?"*

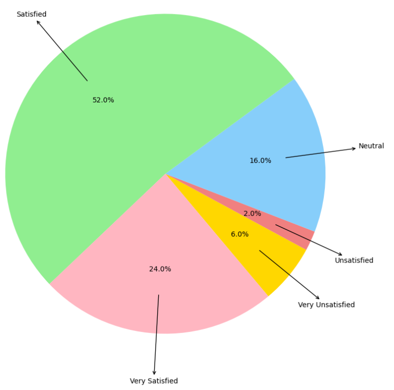


Figure 18. Results for *"How satisfied were you with the Computer Architecture course?"*

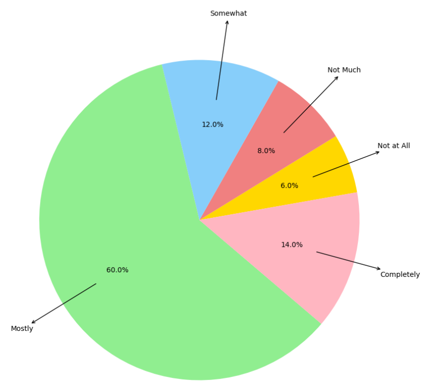


Figure 19. Results for *"To what extent did this course meet your expectations?"*

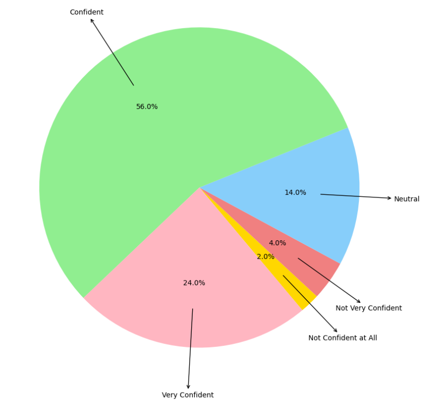


Figure 20. Results for *"How confident do you feel in your understanding of hardware design after completing this course?"*

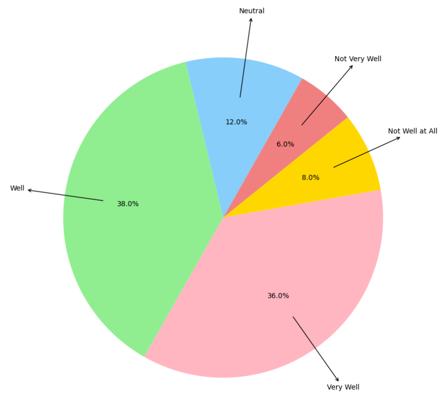


Figure 21. Results for *"How well do you think the course prepared you for low-level system work, such as Computer Hardware Engineering?"*

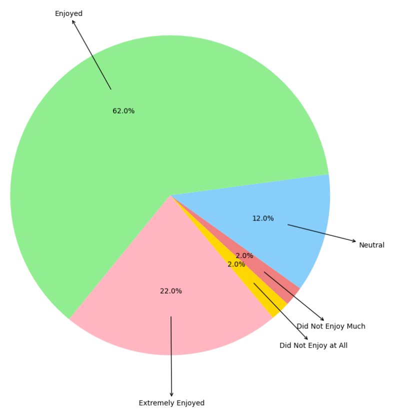


Figure 22. Results for *"How would you describe your overall enjoyment of this course?"*

Table 1. Instruction Format of each Architecture

MuSe Architecture	R-Type	Field	opcode	is_jump	is_imm	rs	rt	rd	unsued	Total
		Bit	3	1	1	3	3	3	2	16
	I-Type	Field	opcode	is_jump	is_imm	rs	rt	immediate		Total
		Bit	3	1	1	3	3	5		16
	J-Type	Field	opcode	is_jump	is_imm	label				Total
		Bit	3	1	1	11				16
DoMe Architecture	R-Type	Field	opcode	control_bit	rt	rs	function code			Total
		Bit	4	1	3	3	5			16
	I-Type	Field	opcode	control_bit	rt	immediate				Total
		Bit	4	1	3	8				16

Table 2. Questions and Results from Survey

No	Question	Avg. Res.	Std. Dev.
1	How would you rate your knowledge and experience level about hardware design before/after the course?	3.12/7.02	2.71/2.14
2	How would you rate your self-learning ability before/after the course?	4.64/7.52	1.99/1.93
3	How interested were you in working on low-level systems, such as Computer Hardware Engineering before/after the course?	3.92/4.92	2.50/2.89
4	How aware were you of developing software by considering hardware abilities and limitations before the course?	6.98	2.13
5	How did the use of visual tools during the course impact your learning experience?	3.84	1.04
6	How did working in groups during the course affect your learning experience?	3.46	1.20
7	How did hands-on experiences during the course influence your learning experience?	4.12	0.99
8	How did implementing and designing your own architecture during the course affect your learning experience?	3.78	1.02
9	How did using simulators and tools shared on the internet during the course impact your learning experience?	4.04	0.82
10	How satisfied were you with the Computer Architecture course?	3.86	1.00
11	To what extent did this course meet your expectations?	3.68	1.00
12	How confident do you feel in your understanding of hardware design after completing this course?	3.96	0.84
13	How well do you think the course prepared you for low-level system work, such as Computer Hardware Engineering?	3.88	1.19
14	How would you describe your overall enjoyment of this course?	4.00	0.77

Table 3. Instruction List (**Not a MUST instruction*)

		000	0	0	rs	rt	rd	unused	ADD
		001	0	0	rs	rt	rd	unused	SUB
		100	0	0	rs	rt	rd	unused	MUL
		010	0	0	rs	rt	rd	unused	AND
	R-Type	011	0	0	rs	rt	rd	unused	OR
		101	0	0	rs	rt	rd	unused	SLL
		110	0	0	rs	rt	rd	unused	SRL
		101	1	0	rs	rt	rd	unused	*SYSCALL
		111	0	0	rs	rt	rd	unused	SLT
MuSe Architecture		001	1	0	rs	rt	rd	unused	JR
		101	0	1	rs	rt	imm[4:0]		LUI
		111	0	1	rs	rt	imm[4:0]		SLTI
		100	0	1	rs	rt	imm[4:0]		MULI
	I-Type	001	0	1	rs	rt	imm[4:0]		BEQ
		011	0	1	rs	rt	imm[4:0]		BNE
		000	0	1	rs	rt	imm[4:0]		SW
		010	0	1	rs	rt	imm[4:0]		LW
		000	1	0	imm[10:0]				JAL
	J-Type	001	1	0	imm[10:0]				J
		1000	control_bit	rt	rs	00010			AND(C)
		1000	control_bit	rt	rs	01000			OR(C)
		1000	control_bit	rt	rs	00000			ADD(C)
		1000	control_bit	rt	rs	01101			SUB(C)
		1000	control_bit	rt	rs	01010			SLT(C)
		1000	control_bit	rt	rs	00110			SRL(C)
		1000	control_bit	rt	rs	00101			MUL(C)
	R-Type	1000	control_bit	rt	rs	01011			SLL(C)
		1000	control_bit	rt	rs	11010			*SLLV(C)
		1000	control_bit	rt	rs	11011			*XOR(C)
		1000	control_bit	rt	rs	10110			*SRLV(C)
		1000	control_bit	rt	rs	10111			*SRAV(C)
		1000	control_bit	rt	rs	11111			*DIV(C)
DoMe Architecture		1110	1	rt	imm[7:0]				LUI
		1100	1	rt	imm[7:0]				SLTI
		1101	1	rt	imm[7:0]				MULI
		0000	0	rt	imm[7:0]				BEQ
		0000	1	rt	imm[7:0]				BNE
		1111	1	rt	imm[7:0]				LW
	I-Type	0011	1	rt	imm[7:0]				SW
		0001	0	rt	imm[7:0]				J
		0001	1	rt	imm[7:0]				JR
		1001	1	rt	imm[7:0]				JAL
		0101	1	rt	imm[7:0]				*SRA
		1011	1	rt	imm[7:0]				*ADDI

Table 4. Operation Control Values

Group Name	Operation	Operation Control			
MuSe Architecture	add	0	0	0	-
	sub	0	0	1	-
	and	0	1	0	-
	or	0	1	1	-
	mul	1	0	0	-
	sll	1	0	1	-
	srl	1	1	0	-
	slt	1	1	1	-
DoMe Architecture	add	0	0	0	0
	sub	0	0	0	1
	and	0	0	1	0
	or	0	0	1	1
	mul	0	1	0	0
	sll	0	1	0	1
	srl	0	1	1	0
	slt	0	1	1	1
	div	1	0	0	0
	xor	1	0	0	1

Table 5. MuSe Architecture Control Signal Values

Format	Instruction	RegDst	ALUSrc	RegWrite	MemRead	MemWrite	Branch	ALUOP	Jump	JumpReg	ShiftReg	Syscall
R	add	1	0	1	0	0	0	0	0	0	0	0
	sub	1	0	1	0	0	0	1	0	0	0	0
	mul	1	0	1	0	0	0	100	0	0	0	0
	and	1	0	1	0	0	0	10	0	0	0	0
	or	1	0	1	0	0	0	11	0	0	0	0
	sll	1	0	1	0	0	0	101	0	0	0	0
	srl	1	0	1	0	0	0	110	0	0	0	0
	jr	1	0	1	0	0	0	xxx	x	1	1	0
	syscall	0	0	0	0	0	0	xxx	x	0	0	1
	slt	1	0	1	0	0	0	100	0	0	0	0
I	lui	0	1	1	0	0	0	101	0	0	1	0
	slti	0	1	1	0	0	0	100	0	0	0	0
	muli	0	1	1	0	0	0	100	0	0	0	0
	beq	0	0	0	0	0	1	1	0	0	0	0
	bne	1	0	0	0	0	1	1	0	0	0	0
	sw	0	1	0	0	1	0	0	0	0	0	0
	lw	0	1	1	1	0	0	0	0	0	0	0
J	jal	0	0	1	0	0	0	xxx	1	0	0	0
	j	0	0	0	0	0	0	xxx	1	0	0	0

Table 6. DoMe Architecture Control Signal Values

Format	Instruction	ALUSrc	RegWrite	MemRead	MemWrite	ALUOp3	ALUOp2	ALUOp1	ALUOp0	Jump	Branch
R	add	0	1	0	0	0	0	0	0	0	0
	sub	0	1	0	0	0	0	0	1	0	0
	mul	0	1	0	0	0	1	0	0	0	0
	and	0	1	0	0	0	0	1	0	0	0
	or	0	1	0	0	0	0	1	1	0	0
	sll	0	1	0	0	0	1	0	1	0	0
	srl	0	1	0	0	0	1	1	0	0	0
	sllv	0	1	0	0	0	1	0	1	0	0
	srlv	0	1	0	0	0	1	1	0	0	0
	div	0	1	0	0	1	0	0	0	0	0
	xor	0	1	0	0	1	0	0	1	0	0
	srav	0	1	0	0	0	1	1	0	0	0
	slt	0	1	0	0	0	1	1	1	0	0
I	lui	1	1	0	0	0	1	0	1	0	0
	slti	1	1	0	0	0	1	0	0	0	0
	muli	1	1	0	0	0	1	0	0	0	0
	beq	0	0	x	0	0	0	0	1	0	1
	bne	0	0	x	0	0	0	0	1	0	1
	lw	1	1	1	0	0	0	0	0	0	0
	sw	1	0	0	1	0	0	0	0	0	0
	sra	1	1	0	0	0	1	1	0	0	0
	addi	1	1	0	0	0	0	0	0	0	0
	jr	x	0	x	0	x	x	x	x	1	x
	jal	x	1	x	0	x	x	x	x	1	x
	j	x	0	x	0	x	x	x	x	1	x

REFERENCES

Anderson, C. L. and Nguyen, M. (2005). A survey of contemporary instructional operating systems for use in undergraduate courses. *Journal of Computing Sciences in Colleges*, 21(1):183–190.

Angelov, V. and Lindenstruth, V. (2009). The educational processor sweet-16. In *2009 International Conference on Field Programmable Logic and Applications*, pages 555–559. IEEE.

Araujo, M. R. D., Padua, F. L. C., Andrade, F. V., and Correa-Junior, F. L. (2014). Mips x-ray: A mars simulator plug-in for teaching computer architecture. *International Journal of Recent Contributions from Engineering, Science & IT (iJES)*, 2(2):36–42.

Aviv, A. J., Mannino, V., Owlarn, T., Shannin, S., Xu, K., and Loo, B. T. (2012). Experiences in teaching an educational user-level operating systems implementation project. *ACM SIGOPS Operating Systems Review*, 46(2):80–86.

Bhagat, S. M. and Bhandari, S. U. (2018). Design and analysis of 16-bit risc processor. In *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, pages 1–4. IEEE.

Binkert, N. L., Dreslinski, R. G., Hsu, L. R., Lim, K. T., Saidi, A. G., and Reinhardt, S. K. (2006). The m5 simulator: Modeling networked systems. *Ieee micro*, 26(4):52–60.

Black, M. (2016). Export to arduino: a tool to teach processor design on real hardware. *Journal of Computing Sciences in Colleges*.

Brox, M., Gersnoviez, A., Montijano, M. A., Herruzo, E., and Moreno, C. D. (2018). Sicome 2.0: A teaching simulator for computer architecture. In *2018 XIII technologies applied to electronics teaching conference (taee)*, pages 1–7. IEEE.

Burch, C. (2002). Logisim: A graphical system for logic circuit design and simulation. *Journal on Educational Resources in Computing (JERIC)*, 2(1):5–16.

Christopher, W. A., Procter, S. J., and Anderson, T. E. (1993). The nachos instructional operating system. In *USENIX Winter*, pages 481–488. Citeseer.

Djordjevic, J., Nikolic, B., and Milenkovic, A. (2005). Flexible web-based educational system for teaching computer architecture and organization. *IEEE Transactions on Education*, 48(2):264–273.

Dogan, M. (2023). Dome risc16 verilog codes.

Dogramaci, S. and Cataltas, M. (2023a). Muse processor simulator codes.

Dogramaci, S. and Cataltas, M. (2023b). Muse processor verilog codes.

Hennessy, J. L. and Patterson, D. A. (2017). *Computer architecture: a quantitative approach*. Elsevier.

Hsu, D. K. (2015). Case studies in operating systems and global marketing. *E-Leader International Journal*, 10(1).

Jaumain, M., Osée, M., Richard, A., Vander Biest, A., and Mathys, P. (2007). Educational simulation of the risc processor. In *International Conference on Engineering Education*.

Kehagias, D. (2016). A survey of assignments in undergraduate computer architecture courses. *International Journal of Emerging Technologies in Learning*, 11(6).

Kurniawan, W. and Ichsan, M. H. H. (2017). Teaching and learning support for computer architecture and organization courses design on computer engineering and computer science for undergraduate: A review. In *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pages 1–6. IEEE.

Leibovitch, C. Y. R. and Levin, I. (2011). Reinforcing and enhancing understanding of students in learning computer architecture. *Navigating Information Challenges*, 8:157.

Likert, R. (1932). A technique for the measurement of attitudes. *Archives of psychology*.

Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. (2002). Simics: A full system simulation platform. *Computer*, 35(2):50–58.

Malliarakis, C., Satratzemi, M., and Xinogalos, S. (2016). Cmx: the effects of an educational mmorpg on learning and teaching computer programming. *IEEE Transactions on Learning Technologies*, 10(2):219–235.

McGrew, T., Schonauer, E., and Jamieson, P. (2019). Framework and tools for undergraduates designing risc-v processors on an fpga in computer architecture education. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 778–781.

Metin, O. and Dogan, M. (2023). Dome cpu16 simulator codes.

MIPS (2001). Mips32 architecture for programmers. vol. ii: The mips32 instruction set.

Morgan, F., Beretta, A., Gallivan, I., Clancy, J., Rousseau, F., George, R., Bakó, L., and Callaly, F. (2021). Risc-v online tutor. In *International Conference on Remote Engineering and Virtual Instrumentation*, pages 131–143. Springer.

Murdocca, M. J. and Heuring, V. P. (1999). *Principles of Computer Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition.

Nayak, A. S., Hiremath, N. D., Umadevi, F., and Garagad, V. G. (2021). A hands-on approach in teaching computer organization & architecture through project based learning. *Journal of Engineering Education Transformations*, 34:742–746.

Nayak, A. S. and Vijayalakshmi, M. (2013). Teaching computer system design and architecture course — an experience. In *2013 IEEE International Conference in MOOC, Innovation and Technology in Education (MITE)*, pages 21–25.

Nikolic, B., Radivojevic, Z., Djordjevic, J., and Milutinovic, V. (2009). A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization. *IEEE Transactions on Education*, 52(4):449–458.

Noone, M. and Mooney, A. (2018). Visual and textual programming languages: a systematic review of the literature. *Journal of Computers in Education*, 5(2):149–174.

Nova, B., Ferreira, J. C., and Araújo, A. (2013). Tool to support computer architecture teaching and learning. In *2013 1st International Conference of the Portuguese Society for Engineering Education (CISPEE)*, pages 1–8.

Omer, U., Farooq, M. S., and Abid, A. (2021). Introductory programming course: review and future implications. *PeerJ Computer Science*, 7:e647.

Patel, S. and Patt, Y. (2019). *Introduction to Computing Systems: from bits gates to C beyond*. McGraw-Hill.

Patterson, D. A. and Hennessy, J. L. (2016). *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann.

Prvan, M. and OžEGOVIć, J. (2020). Methods in teaching computer networks: A literature review. *ACM Trans. Comput. Educ.*, 20(3).

Rao, K. V., Angeline, A. A., and Bhaaskaran, V. K. (2015). Design of a 16 bit risc processor. *Indian Journal of Science and Technology*, 8(20):1.

Ristov, S., Stolikj, M., and Ackovska, N. (2011). Awakening curiosity—hardware education for computer science students. In *2011 Proceedings of the 34th International Convention MIPRO*, pages 1275–1280. IEEE.

Sarder, M. (2014). Improving student engagement in online courses. In *2014 ASEE Annual Conference & Exposition*, pages 24–719.

Schuurman, D. C. (2013). Step-by-step design and simulation of a simple cpu architecture. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 335–340.

Simkins, D. and Decker, A. (2016). Examining the intermediate programmers understanding of the learning process. In *2016 IEEE Frontiers in Education Conference (FIE)*, pages 1–4. IEEE.

Theys, M. D. and Troy, P. A. (2003). Lessons learned from teaching computer architecture to computer science students. In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 2, pages F1C–7. IEEE.

Thomas, N., Carroll, F., Kop, R., and Stocking, S. (2012). ibook learning experience: the challenge of teaching computer architecture to first year university students. In *Proceedings of the International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government (EEE)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer ....

U.S. Bureau of Labor Statistics (2021). Web developers and digital designers : Occupational outlook handbook.

U.S. Bureau of Labor Statistics (2022a). Computer hardware engineers : Occupational outlook handbook.

U.S. Bureau of Labor Statistics (2022b). Information security analysts : Occupational outlook handbook.

Velev, M. (2003). Integrating formal verification into an advanced computer architecture course. In *2003 Annual Conference*, pages 8–737.

Vijayalaskhmi, M. and Karibasappa, K. (2012). Activity based teaching learning in formal languages and automata theory-an experience. In *2012 IEEE International Conference on Engineering Education: Innovative Practices and Future Trends (AICERA)*, pages 1–5. IEEE.

Vollmar, D. K. and Sanderson, D. P. (2005). A mips assembly language simulator designed for education. *J. Comput. Sci. Coll.*, 21(1):95–101.

Vollmar, K. and Sanderson, P. (2006). Mars: an education-oriented mips assembly language simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 239–243.

Wallace, C. S. (1964). A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17.

Wu, H.-T., Hsu, P.-C., Lee, C.-Y., Wang, H.-J., and Sun, C.-K. (2014). The impact of supplementary hands-on practice on learning in introductory computer science course for freshmen. *Computers & Education*, 70:1–8.

Xilinx (2007). Data2mem user guide.

Yehezkel, C., Yurcik, W., Pearson, M., and Armstrong, D. (2001). Three simulator tools for teaching computer architecture: Little man computer, and rtlsim. *Journal on Educational Resources in Computing (JERIC)*, 1(4):60–80.

Yıldız, A., Ugurdag, H. F., Aktemur, B., İskender, D., and Gören, S. (2018). Cpu design simplified. In *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, pages 630–632. IEEE.