Teaching computer architecture by designing and simulating processors from their bits and bytes

Mustafa Doğan [Corresp., 1], **Kasım Öztoprak** [2], **Mehmet Reşit Tolun** [3]

[1] ASELSAN Research, Ankara, Turkey

[2] Department of Computer Engineerig, Konya Food and Agriculture University, Konya, Turkey

[3] Department of Software Engineering, Çankaya University, Ankara, Turkey

Corresponding Author: Mustafa Doğan
Email address: mustafadogan@aselsan.com.tr

Teaching Computer Architecture (CA) course in undergraduate curricula is becoming more of a challenge as most students prefer software-oriented courses. In some computer science/engineering departments, CA courses are offered without the lab component. This paper demonstrates how students working in teams are motivated to study the CA course and how instructors can increase their motivation and knowledge by taking advantage of the hands-on practices described. Firstly, the teams are asked to design and implement their 16-bit MIPS-like processors from the scratch under certain limitations such as using a specified instruction set, and limited data and instruction memory.

Student projects comprised of three phases, namely, design, desktop simulator implementation, and verification using Hardware Description Language (HDL). In the design phase, teams developed their architectures to implement specified instructions. Although teams' requirements and limitations are the same, their designs differed. For example, while one of the teams designed a processor with extensive features by having user-defined instructions resulting in longer cycle times, the other team preferred to build a processor with a minimal instruction set but with a faster clock cycle time. Next, teams developed a desktop simulator in any of the programming languages to execute instructions on the architecture they built. Lastly, they created HDL projects in Verilog to verify their data-path realized in the design phase by executing instructions in the ISim simulator environment supported by ISE. At the completion stage of the project, students' thoughts and their current understanding of the project are collected by means of a questionnaire using a ten-point Likert scale. Results of the survey show that the hands-on approach increase students' motivation and knowledge in the CA course and this approach can be extended to similar courses, ie. Microprocessor Design, with some minor changes. Furthermore, the present study demonstrates that interactions between students in each phase increased their potential knowledge and perspective on designing custom processors.

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

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

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

Corresponding author:
Mustafa Dogan[1]

Email address: mustafadogan@aselsan.com.tr

ABSTRACT

Teaching Computer Architecture (CA) course in undergraduate curricula is becoming more of a challenge as most students prefer software-oriented courses. In some computer science/engineering departments, CA courses are offered without the lab component. This paper demonstrates how students working in teams are motivated to study the CA course and how instructors can increase their motivation and knowledge by taking advantage of the hands-on practices described. Firstly, the teams are asked to design and implement their 16-bit MIPS-like processors from the scratch under certain limitations such as using a specified instruction set, and limited data and instruction memory.
Student projects comprised of three phases, namely, design, desktop simulator implementation, and verification using Hardware Description Language (HDL). In the design phase, teams developed their architectures to implement specified instructions. Although teams' requirements and limitations are the same, their designs differed. For example, while one of the teams designed a processor with extensive features by having user-defined instructions resulting in longer cycle times, the other team preferred to build a processor with a minimal instruction set but with a faster clock cycle time. Next, teams developed a desktop simulator in any of the programming languages to execute instructions on the architecture they built. Lastly, they created HDL projects in Verilog to verify their data-path realized in the design phase by executing instructions in the ISim simulator environment supported by ISE. At the completion stage of the project, students' thoughts and their current understanding of the project are collected by means of a questionnaire using a ten-point Likert scale. Results of the survey show that the hands-on approach increase students' motivation and knowledge in the CA course and this approach can be extended to similar courses, ie. Microprocessor Design, with some minor changes. Furthermore, the present study demonstrates that interactions between students in each phase increased their potential knowledge and perspective on designing custom processors.

1 INTRODUCTION

As computing technology evolves at an immense pace, it becomes harder to go back to the origins of hardware in order to understand its internal workings. In every improvement either in hardware or software technology, scientists and engineers try to hide technical details from the users. This obviously requires a strong computer engineering background and understanding. The CA course offered in the 3rd year of the Computer Engineering curriculum at Konya Food & Agriculture University (hereafter referred to as KFAU) is one of the essential courses that conveys the necessary background knowledge with hands-on experience to undergraduate students. As expected, when it comes to teaching CA to undergrad students, the task becomes more challenging since most of the students are not really interested in learning computer hardware details because the nature of the CA course looks too theoretical in their eyes and requires an understanding of the internal mechanisms of the computer that they think it is very difficult to grasp.

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 shows us the attention of students and fresh graduates shifts to recent developments and emerging job vacancies in different fields of computing regardless of their salary.

Unfortunately, for students' this situation can be not only effective in CA course, but also in other must courses in the curriculum. According to the KFAU Computer Engineering curricula, students take common core courses in faculty throughout the first three semesters. In the remaining five semesters where students gain computer engineering skills, there are 16 must course and students suffer from the problem given above in 4 different courses which are *Programming Languages, Computer Architecture, Operating Systems, Automata Theory and Formal Languages* (Noone and Mooney, 2018; Thomas et al., 2012; Anderson and Nguyen, 2005; Vijayalaskhmi and Karibasappa, 2012). As the number of theoretical and requiring low-level computer understanding courses increases, which is approximately 25% of the curriculum in KFAU, we believe they can make a huge impact on students' career paths. Furthermore, current studies fail to propose a practical and quantifiable approach to be used for eliminating these problems in CA course.

This study creates the following assignments for the CA course and analyzed these assignments contributions to students and the way how knowledge diffusion among students can increase.

- processor design shown in Fig. 1 including Instruction Set Architecture (ISA) and datapath.
- a desktop simulator demonstrating how the processor executes given code and how contents of memory and registers changed throughout the execution process.
- verifying the designed processor by using HDL. At the end of the project, students take a survey about their thoughts and assessments about the conduct of the course.

The approach offered in this study can be applied to other courses with minor changes and increase both the students' motivation and potential knowledge of these theoretical courses. Besides, if the requirements and limits of assignments are arranged well, students would be more creative and focus on different aspects of user needs. Hence students are able to visualize and examine various scenarios and use-cases in a course. Furthermore, this approach would be beneficial in classes where student participation is relatively low, especially in online or distance education classes, and be a good assessment tool for teachers while grading their students' efforts. Instructors can access and replicate this work shared on GitHub and YouTube, with small modifications to be used in their CA classes.[1]

This study, we believe for the first time in the literature, proposes an approach to increase students' interest, motivation, and knowledge in CA course by examining two different student groups who design their own processors under certain instruction and hardware limitations, and the techniques they used in this process and the effects of these techniques on their designs and knowledge. Besides, this study incorporates detailed documentation that can be traceable and replicable by others.

The rest of the paper is organized as follows: The background, design phase, simulator implementation phase, and hardware validation by Verilog phase is described in sections 2, 3, 4, and 5 respectively. In the next section, the results section, data collected from students using surveys is analyzed. In the discussion part where it is mentioned the strengths and weaknesses of the project are described. Finally, the paper is concluded with the conclusions section and future work.

[1] Playlist for MuSc and DoMe Architecture projects

2 RELATED WORK

In the past, there were many scientific studies conducted on the difficulties encountered in CA 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 CA course.

Thomas et al., 2012 remark on what kind of difficulties and problems students can experience in a CA 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 course 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 CA course due to the fact that CA course are comprised of different contiguous fields, such as digital design, embedded systems, operating systems, etc., and make them complex (Leibovitch and Levin, 2011). On the other hand, Patel and Patt, 2019 state the main reason is 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 CA course (Kehagias, 2016). Kehagias, 2016 survey given every single assignment including basic question answers and sophisticated programming assignments, in CA course at top 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 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). Morgan et al., 2021 developed RISC-V Online Tutor which can also be used in CA courses (Morgan et al., 2021). Wu et al., 2014 state that students who utilized hands-on practices have significantly higher scores than the one who do not in an introduction to computer science course (Wu et al., 2014). Furthermore, hands-on practices decreased students' stress levels during the course.

Nikolic et al., 2009 survey and evaluate the current simulators which are used to teach CA 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). Many available simulators cover about one-third of the course content, whereas M5 and Simics cover around two-thirds. 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. 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 made a difference among those studies and developed a simulation where students enter the assembly instructions as input and track each electric signal step by step (Jaumain et al., 2007).

Rao et al., 2015 came up with such a processor design that all components such as ALU, control unit, instruction decoder, etc. are carefully selected to achieve better performance (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 propose a soft CPU simulation platform called VerySimpleCPU (VSCPU) to allow students to design their processors from the scratch and build code for their processors and implement it using FPGA easily (Yıldız et al., 2018). This tool concentrates on making the processor design easier for the students, rather than teaching them to understand and design a processor with its bits and bytes.

3 THE PROCESSOR (CPU) DESIGN

This section, defines firstly, the fundamental design limitations each processor must support. Then, it analyzes and compares the differences of each processor design. Before examining the processor design, it will be appropriate to give some insight into students' knowledge prior to taking the CA course. Students had a mathematical background, digital design, and basic Verilog knowledge before starting on the CPU design as they have taken Programming (Java, Python, C/C++), Discrete Mathematics, and Logic Design courses previously. Although their programming abilities are sufficient to carry out desktop simulator development, as students referred to in the questionnaire, they faced challenges during the design phase using HDL due to their lack of HDL programming experience.

In this phase of the project, students specified their instruction architecture, data-path, control signals, supported instruction list, and arithmetic logic unit design. Students had two weeks to come up with their architectural ideas. As Omer et al., 2021 suggest to instructors, we attached significant importance to collaborative learning and therefore each group made a presentation of what they have done in each phase (Omer et al., 2021). In these presentations, students discuss and criticize their friends' designs and others are responsible to clarify what they have done and why. After the discussion part, students are allowed to make changes in their designs to achieve a better version of it by replacing strong aspects of their friends' designs with their weak ones.

At the beginning course, we planned to have an extra phase for the project regarding the physical implementation of the designs made by students. We believe that this phase would be an important factor for their knowledge. However, due to the COVID-19 pandemic, the university and its facilities were closed during the semester and this phase of the project had to be canceled in the middle of the semester. Obviously, this unexpected situation caused students to reconsider their design. More details about students' thoughts and the effects of this phase on the processor design are discussed in the following sections. If this phase had been carried out, students would be responsible to build their processors using transistor-transistor logic which meant they should have used various integrated circuits, timers, and breadboards. In addition, students should have used memory components to store instructions and data, and an LCD screen to visualize registers.

Since there are two different CPU architectures from two different groups, same indicators

are used to differentiate between them, namely, MuSe and DoMe Architectures. Both architectures used MIPS architecture as a starting point and built their architecture on top of it (MIPS, 2001).

3.1 Prerequisites of CPUs

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

The specifications of CPUs had to be carefully selected because eventually it is expected that such a CPU is simple enough but allows students to see the general picture and has the minimal ability to contain instructions of a generic processor, for instance, a 16-bit CPU. The reason behind designing a 16-bit CPU rather than an 8-bit or 32-bit is that an 8-bit CPU design would be its easiness for a course project, but a 32-bit CPU design could not have been completed on time due to project limitations.

3.2 MuSe Architecture: The Processor Design Stage

Here all the key elements of MuSe Architecture, including Multiplication Algorithm, Instruction Set & Format, data-path, and lastly ALU design are described. Students constituted detailed work on ALU design, data-path, and control signals for their architecture (additional details are explained in the supplementary). The MuSe Architecture is designed in such a way that it supports and conforms to all requirements mentioned in section 3.1 at the same time, taking care of performance issues which are encountered in the DoMe Architecture.

3.2.1 MuSe Architecture: The Multiplication Algorithm

Multiplication is one of the time-consuming instructions that is performed in the ALU. In MuSe Architecture, designers used the multiplication algorithm proposed by Patterson and Hennessy, 2016 to implement a multiplication method iteratively by using registers and adders as shown in Fig. 2. Since an iterative approach is not feasible for a single-cycle data-path, MuSe Architecture is designed with a multiplication unit in the ALU, that was composed of eight shift registers and eight 8-bit adders.

3.2.2 MuSe Architecture: The Instruction Set and Format

While designing a CPU, MuSe Architecture designers split instructions into types just like in the MIPS architecture. In the MIPS architecture, instructions are split into three types R-type, I-type, and J-type instructions. Each instruction type has different fields to perform its tasks (MIPS, 2001). These instructions are distinguished from each other by their operation code (opcode) fields which means each instruction has a unique opcode value, except for R-type instructions. R-type instructions do not include a target address, branch displacement, or immediate field. They have fields for three registers, function code, and shift amounts, unlike other instruction types. To increase functionality, MIPS architecture designers set the opcode of each R-type instruction 0. Function Code field is used to recognize R-type instructions from each other.

In I-type instructions, bits used for function code are used for immediate value. This immediate value is used for the followings: *i)* a constant operand *ii)* a branch target offset *iii)* a memory operand displacement. I-type instructions help users not to use a register for a constant value. J-type instructions are for jumping instructions. They change the flow of a program.

The instruction format that the MuSe Architecture designers proposes is shown in Table 1 focusing on supporting the mandatory instructions with better performance. As in the MIPS architecture, MuSe Architecture designers decided to have instructions with three types: R, I, and J. While deciding on fields in the instruction formats, they aimed to utilize the 3-bit opcode

field as ALU operation code to reduce the decoding complexity ~~of instruction~~. The idea is to first look at ***is_jump*** and ***is_imm*** fields and decode instruction accordingly. In the decoding phase, they ~~tried to~~ choose 3-bit opcodes with the same as or closer to ALU operation codes to decrease time and hardware complexity to resolve which operation to select in ALU. Rather than distinguishing R-type instruction from the others by using opcode, MuSe Architecture designers ~~preferred~~ added extra fields to determine the type of instruction called ***is_imm*** and ***is_jump***, ~~which are 1-bit length~~. R-type instructions have three register values.

I-type ~~instructions utilize~~ an immediate value. This type of instruction does not have an ***Rd*** field. Yet, they include a 5-bit immediate field for storing an immediate value. Because I-type instructions use immediate values, ***is_imm*** field is always set to 1.

J-format ~~instructions are used for~~ jump instructions. ~~Instructions in this category~~ have no ***Rd***, ***Rs***, and ***Rt*** fields. Instead, they include an 11-bit label field for storing the target address. Because they are intended for jump, ***is_jump*** field is always set to 1.

All types of instructions supported by MuSe Architecture designers are shown in supplementary.

3.2.3 MuSe Architecture: The ALU Design

The Arithmetic Logic Unit (ALU) is the core part of the CPU that performs all calculations. MuSe Architecture ~~handles ALU, and its operations~~ using two input ports ~~called input~~ A and ~~input~~ B, and an output port ~~which is called output~~ C ~~as seen in~~ (Fig. 3). ~~ALU is capable of handling various calculations and~~ type calculation is determined by ALU control lines which are 3-bits (F0, F1, F2) in length. These lines are controlled by the ALUOp control signal introduced in data-path and change whenever an upcoming instruction is decoded. More details about MuSe Architecture ALU capabilities can be found in supplementary.

3.2.4 MuSe Architecture: The Data-path and Control Signals

Data-path ~~is the glue connecting all necessary components of~~ hardware ~~that consists of functional units of processors~~ such as ALU, adders, memory, registers, etc. Moreover, the data-path has control signals which enable different units of CPU. For example, the memory read control signal is set to 1, and it allows us to read from the memory when the load word (*lw*) instruction is called. While creating the data-path, MuSe Architecture designers were inspired by the original MIPS data-path, as shown in Fig. 4. In traditional MIPS architecture, there are eight different control signals (MIPS, 2001). On the other hand, the design of MuSe Architecture has 11 different control signals in order to handle instructions. The additional control signals in addition to the ones in traditional MIPS architecture are System call (***syscall***), Jump Register (***JumReg***), and Shift Register (***Shift Reg***) control signals. In the supplementary, control signal values for each instruction are shared for those who want to replicate MuSe Architecture.

3.3 DoMe Architecture: The Processor Design Stage

This section ~~is about how~~ the DoMe Architecture is designed, and how it differs from the MuSe Architecture. Unlike MuSe Architecture, DoMe Architecture focuses on the more instruction capability. DoMe Architecture designers prefer to have more instructions than mandatory instructions, such as division to make their CPU design more functional. However, this approach takes us a bit far from the spirit of MIPS. On the other hand, it increases the abilities of the processor with some performance degradation as well as allowing the students to explore the diversity.

3.3.1 DoMe Architecture: The Multiplication Algorithm

Here we discuss the DoMe Architecture multiplication algorithm which is handled in the design phase. After extensive research, DoMe Architecture designers decided to use the Wallace Tree multiplication method (Wallace, 1964). However, in the design phase presentation, their project co-advisor suggested using a different multiplication approach since implementing an 8-bit Wallace Tree multiplier is costly and hard to implement in a laboratory environment. Since the implementation of an 8-bit Wallace Tree multiplier requires more than 100 gates, it would be tough for students to implement it. Hence DoMe Architecture designers decided to use a multiplication algorithm suggested by the MuSe Architecture in section 3.2.1.

3.3.2 DoMe Architecture: The Instruction Set and Format

While designing the CPU, DoMe Architecture designers were also inspired by traditional MIPS architecture just like MuSe Architecture, since the MIPS architecture is simple and easy to use (MIPS, 2001). Table 1 shows the instruction format of the DoMe Architecture. DoMe Architecture's instruction format differs in certain ways from MuSe Architecture ~~as given below:~~ 1) ~~DoMe Architecture's~~ instruction format uses a 5-bit function code. 2) has additionally a 1-bit ***control_bit***. 3) does not have a register address for the destination register (***Rd***). 4) uses an 8-bit immediate ~~value while~~ MuSe Architecture uses a 5-bit value) 5) does not allocate extra bits for ***is_imm*** and ***is_jump fields*** fields. 6) does not use an extra type for jumping instructions.

In R-type instruction format, the most significant 4-bits represent opcode, ~~and it~~ determines the type of instruction. If it is an R-type instruction, then the CPU determines instruction according to the function code. Otherwise, it determines the instruction according to the opcode, just like in the MIPS. Following 1 bit is for specifying the resulting register. Next, 6 bits for specifying the source and target registers. Since there are eight general-purpose registers, it can be represented in 3-bits. The least significant five bits are for function code to determine R-type instructions. In I-type instruction format, the first three fields are the same as R-type. However, there is an immediate field in I-type instruction format instead of source register and function code fields.

The DoMe Architecture shares common features with the MIPS architecture, such as the existence of function code. However, there left only two bits to the function code field which is inadequate to support more instruction with a single opcode. Although this approach is not a problem, to increase diversity in the projects the approach of the students is supported by the instructor. Therefore, students propose reducing one of the register addresses in instruction format to gain more bits in the function code field. DoMe Architecture prefers using a permanent, resulting register among eight general-purpose registers instead of letting the user determine the register. In other words, if a user wants to make an addition operation with the values of *r1*, and *r2* registers, and store the summation of these values in the *r3* register, a user first should make an addition operation using *r1* and *r2*. As a result of the addition operation, the CPU stores the result in a default register called ***Rd***. Eventually, the user needs to move the resulting value in ***Rd*** to the intended register. Furthermore, DoMe Architecture proposed another result storage approach for users. According to this approach, a user will be able to store the result in the target register using the ***control_bit*** specified in Fig. 5 which helps the user to select one of these storage approaches. If a user wants to store the result in the destination register, then the ***control_bit*** must be set to 1, otherwise 0. DoMe Architecture designers provide the "*-c*" suffix to their instruction set to distinguish where the result is stored. This issue only applies to R-type instruction and do not support I-type instructions since I-type instructions use one register.

The way of thinking behind the lack of a J-type instruction set is that the instruction memory is limited to 256 Byte and DoMe Architecture's I-type format has ~~an 8-bits~~ immediate part which makes them reach any instruction in the instruction memory. Furthermore, having an 8-bits immediate part rather than 5-bits helps users to make calculations with constants larger than 32 ~~easier~~. More details about instructions in DoMe Architecture can be found in the supplementary.

3.3.3 DoMe Architecture: The ALU Design

DoMe Architecture is designed to support more instructions than MuSe Architecture. Therefore DoMe Architecture instructions contain division and exclusive or (***xor***) operations besides the other eight operations. Since it is not possible to represent 10 different operations with three control lines, DoMe Architecture designers added one more control line (F3), as shown in Fig. 3 to their design and left the rest of the six signals unused. More details regarding ALU operations and their corresponding control line values can be found in supplementary.

3.3.4 DoMe Architecture: The Data-path & Control Signals

In this section, we review DoMe Architecture's approach to designing a data-path and control signals. As stated ~~earlier in~~ section 3.2.4, there are eight different control signals in the traditional MIPS architecture, and with MuSe Architecture this has increased to 11. On the other hand, DoMe Architecture decreased control signals to seven, by eliminating Register

Destination (RegDst) and Memory to Register (MemtoReg) control signals. Students used an extra control signal, which is called jump, to adjust jumping instructions. All unit and control signal connections in MIPS data-path are revised accordingly to support DoMe architecture and the resulting data-path is depicted in Fig. 6.

More about control signal values to replicate DoMe Architecture are provided in the supplementary.

4 THE SIMULATOR

CA is one of the complex courses that is enriched by topics from other fields of computer science, such as operating systems, programming languages, etc (Leibovitch and Levin, 2011). To make sure students get the most benefit, this course is often conducted with lab sections where they can gain practical experience (Nikolic et al., 2009). Most of the activities performed in CA course lab can be easily completed using simulators. That's why simulators are one of the best practices that teaching CA course ~~to a person~~ (Burch, 2002; Djordjevic et al., 2005; Vollmar and Sanderson, 2006). Therefore, at different stages of the project, students are asked to create and present their own simulators which must support the following features: 1) a section in which the user can give input 2) ability to parse given assembly code into machine code ~~consistent with processor design~~ 3) ability to visualize current values of registers and memory cells 4) ability to interpret code either step by step or fully automatically 5) students are free to use whichever programming languages and tools they want.

Araujo et al., 2014 propose visualization of the MIPS data-path, MIPS X-Ray, rather than simulating the full system. However, since students will experience this in the HDL design phase, they are expected to create a simulator ~~that looks like the example simulator diagram depicted~~ in Fig. 7 (Wikipedia Contributors, 2022). ~~The user interface~~ controls and visualizes the simulator and it is expected to comprise 4 main units. Instruction parser gets ~~and decodes~~ assembly instructions supported by students' architecture from a user via a user interface. After decoding, the instruction parser forwards the output to the control manager. The control manager manages the ALU manager and register & memory manager. It also provides the flow between these managers. ALU manager is responsible from perform arithmetic calculations. On the other hand, the register & memory manager stores the contents of registers and memory and delivers these data to the user interface.

Before the design phase, students were requested to develop a MIPS simulator that runs with 32-bit MIPS instructions and supports all features of the original MIPS processor (MIPS, 2001). This simulator helped students to understand the MIPS architecture better and have a programming background for future simulators. There are two primary reasons for assigning this task: a) it is planned to let students develop desktop simulators for their architectures from this MIPS simulator. Hence students have more time in the design and verification phase b) MIPS is an easy architecture to understand and would make it easier to design processors from scratch for students. Students had three weeks to develop a simulator for their architectures.

In the following section, differences between simulators and the effect of public presentations on simulator design are analyzed.

4.1 MuSe Architecture: The Simulator

MuSe Architecture designers used Java programming language to develop the simulator and JavaFX for the graphical user interface. The ~~plain~~ simulator shows data memory ~~in a single section that contains the~~ address and value in that address in binary format. ~~There is an extra section for~~ instruction memory ~~which is similar to data memory to~~ visualize the machine code. In the register visualization part, the simulator shows the current register names and their values in a signed decimal format. Additionally, MuSe Architecture designers arranged a section to display the status of control signals in current instructions. When an instruction is being executed, the necessary control signals are highlighted. Furthermore, the current program counter and an LCD Display showing opcode, jump, and immediate value of current instruction are set (Fig. 8). MuSe Simulator designers added an LCD Display section to their simulator due to the existence of LCD Display in physical implementation part. Since registers are already

visualized, students preferred to design the LCD Display section as a software tool showing details about instruction currently being executed in their simulator.

MuSe simulator is composed of several classes which are ALU, Controller Unit, Instructions, Instruction Memory, Data Memory, Register File, Program Counter, and Processor. In ALU class, designers preferred to use simple operators for calculations rather than implementing logic circuits such as full adders and multiplication logic. The Controller Unit class is responsible for assigning control signals of each instruction by comparing opcodes, ***is_jump***, and ***is_imm*** values. MuSe simulator is driven by three particular instruction types which are R-type, I-type, and J-type instructions. Since all these instructions share common attributes such as opcode, registers, etc., the MuSe simulator extends these instructions from an abstract class called Instruction. Instruction Memory is composed of an array of instructions and initialized as the program starts. Data Memory stores a two-dimensional byte array with a stack pointer. All necessary read and write operations in memory are handled in this class. Register File class stores a list of registers with eight pre-defined registers. Register read and write operations are handled in this class according to control signals. Program Counter is such a simple class that stores only an integer value for the program counter value and is responsible for manipulating this value. The Processor class uses an object of previously mentioned classes and is responsible for managing and organizing the workflow of these modules ~~in a harmony~~. As program starts, instruction memory is loaded and the processor starts to fetch instructions in a loop. Since there is no pipeline implementation in MuSe architecture, unless current instruction is operated successfully, the simulator does not progress to the next instruction. The program is terminated ~~whether~~ all instructions are executed or the simulator encounters an error. Students and tutors could ~~easily~~ analyze the architecture of design using the simulator and can change as they wish.[1]

~~In section 3, it is mentioned that~~ each student group makes a presentation that also informs other groups about their project. These presentations contributed to MuSe Architecture in simulator design as well. While creating the MIPS Simulator ~~which is mentioned in Section 4, MuSe~~ Architecture designers did not use control signals and LCD display section in their simulator. They enhanced their MIPS simulator by adding new features while converting it to a simulator that supports their architecture.

4.2 DoMe Architecture: The Simulator

Although MuSe Architecture designers did an excellent job in creating a plain graphical user interface, DoMe Simulator focused more on the representation of data in both memory cells and registers rather than creating a plain interface, which can be seen in Fig. 9. DoMe Simulator ~~used~~ Python programming language ~~for general architecture~~ and ~~used~~ PyQt5 for the graphical user interface. The most significant advantage of the DoMe Simulator is the visualization of both memory and stack. Users can examine and visualize changes in both the memory and stack at the same time. Furthermore, the representation of the memory cells in the DoMe Simulator is not only in binary form but also in decimal format. Also, DoMe Architecture designers preferred more than one way to represent data in the registers. The data is visualized in hexadecimal, signed integer, and unsigned integer forms, making it easier to analyze. On the other hand, DoMe Simulator lacks visualization of control signals and the LCD display section mentioned in MuSe Architecture Simulator in the previous section.

DoMe Simulator has a bunch of classes, some of which are similar to the ones used in the MuSe simulator. DoMe Simulator uses the following classes: Registers, Instructions, Data Memory, Instruction Memory, Definitions, Instruction Functions, Assembler, and Processor. Registers, Instructions, Data Memory, Instruction Memory, and Processor. These classes have been designed with the same approach used in MuSe Simulator which results in that they have the same function. Instructions are stored in an array and fetched by the Processor class in a loop accordingly. Registers are also stored in an array and are initialized here. Data Memory is responsible for memory operations and memory in this class is stored in an array as well. Unlike MuSe Simulator, DoMe Simulator does not have an ALU class. Instead of using a class that simulates the ALU in a data path, they preferred to use such a class, Instruction Functions, that defines a function for each instruction operation and creates a link between function

[1] MuSe Architecture Simulator

and instruction. The simulator directly accesses these functions and executes the necessary operation. Assembler class is a variant of Control Unit class in MuSe simulator. It assigns control bits, opcodes, registers, and other values. Definitions class is a ~~kind of~~ look-up table that pre-defines the properties of each instruction. It is used by the Assembler class to set instruction attributes. Overall, although DoMe Simulator does not implement exact components ~~in their data path~~, they are aware of this part of the project focuses on more capability and workflow of their architecture. Students and instructors could utilize this simulator by not only using and replicating the simulator but also contributing to this open-source project by adding new functionalities.[1]

5 VERIFYING THE CPU DESIGN USING A HARDWARE DESCRIPTION LANGUAGE

~~Up until now~~, students learned how CA is designed and its instructions are handled ~~theoretically~~. ~~At this point~~, students ~~have~~ completed the processor's design with the given ISA including the CPU, ALU, instruction format, and data path. ~~In the next step~~, the simulator ~~will be used~~ to demonstrate the execution of the code. ~~Before implementing the design in a laboratory environment, it should be verified at a hardware level using an Hardware Description Language (HDL). So, students focus more on hardware practices~~. Students will create an HDL project ~~using~~ their architecture. ~~HDL projects must simulate students' data paths to ensure that architecture works fine.~~ However, students are free to use some conveniences provided by IDE, such as ALU operations. This phase of the project lasted four weeks ~~which is the longest phase~~ because students' HDL background was at the elementary level ~~and they needed time to practice as mentioned in section 3.~~ In this phase of CPU design, groups ~~preferred to~~ used the Verilog hardware description language and implemented their HDL code ~~in~~ the Xilinx ISE Design Suite development environment (Xilinx, 2007). The Xilinx ISE Design Suite provides users with a simulation application which is called ISim. Users can examine their designs step by step and module by module in ISim. Although Xilinx provides a new development environment called Vivado and no longer supports ISE Design Suite, ~~since~~ the ISE Design Suite requires fewer System Requirements ~~and enough to simulate designs; groups decided to use ISE Design Suite instead of Vivado. There is no ISE project template given to students and they are expected to implement their own projects.~~

~~Students are asked to implement~~ multiplication operation ~~in their CPU design. There are different methods and ways to implement multiplication operand, especially while implementing in Verilog~~. Both groups preferred to use the multiplication method proposed by Patterson and Hennessy, 2016 which deals with negative values by converting the negative values to positive and deciding the sign bit at the end ~~as shown in~~ Fig. 2. Since students make Verilog implementation for validating the hardware design, MuSe Architecture designers implemented this multiplication method in Verilog implementation part as well. ~~However,~~ DoMe Architecture designers chose to implement this instruction using the "*" operand in Verilog implementation to avoid potential hazards during the simulation.

~~In Verilog, the projects comprise modules. A module is a piece of Verilog code that performs a specific task. Modules can be embedded into other modules, and a higher-level module's input and output ports can be used to connect with its lower-level modules~~. In the following sections, the modules used by each architecture are described.

5.1 MuSe Architecture: The Verilog Design

Verilog design of MuSe Architecture shown in Fig. 10 utilizes eight different Verilog modules: a) Instruction Memory module containing instructions fed by a user in binary format. The module takes the program counter as the input and returns the instruction which is going to be executed. b) Data Memory module helps us to control data in the memory. This module takes data, addresses, and some control signals as input and returns data read from memory. c) Register File module contains registers, and users can perform changes on registers using this module. d) ALU module performs almost all operations such as summation, subtraction, multiplication, address calculation, etc. As described in the data-path, the ALU module takes two source inputs and one ALU control lines. According to these inputs, the ALU module carries

[1] DoMe Architecture Simulator

out necessary calculations and returns an output value. e) Control Unit module takes opcode, ***is_jump***, ***is_imm*** values as input, and it updates all control signals accordingly. f) The processor module is the main module that calls other modules and makes them work in harmony. g) The LCD module is a module that helps users to examine the content of registers and memory cells.

5.2 DoMe Architecture: The Verilog Design

DoMe Architecture's Verilog design shown in Fig. 11 utilizes more Verilog modules than MuSe Architecture. 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) GPRs module describes eight general-purpose registers and stores the values. This module helps us to read and write in registers. c) 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) ALU Control module determines the ALU Control Lines values for the specified instruction. It uses opcode, function code, and ALU operation code to decode ALU control lines. f) 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. g) Data-path unit manages the program counter's updating process and in-out operations from either memory or register. This module is a unit where almost all process is done. h) 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.

Students shared their Verilog codes on GitHub.[1, 2]

6 RESULTS

To achieve quantitative results about students' experiences and thoughts during the project, 21 students answer a ten-point Likert scale questionnaire (Likert, 1932) whose questions and average results are shown in Table 2, respectively. Students took this questionnaire twice, one at the beginning and the other at the end of the semester. Students are fed with random questions to avoid order effects bias. Furthermore, students had a fair amount of time, in this case, one hour, to complete the questionnaire and avoid any procedural bias. Lastly, questions are made as clear as possible and neutral to avoid leading question bias.

The results of the questionnaire show that students had a hardware design knowledge of 4.33 points on average and after taking the CA course with the approach presented in the paper, this knowledge has risen to 7 points. Moreover, students state that if they were exposed to an ordinary recitation or documents and tools shared online rather than a 3-phased project approach, their hardware design knowledge would be limited to 5.28 and 6.14 on average, respectively. Furthermore, results indicate that the hands-on practices and implementing designs from the scratch have significantly increased the knowledge obtained in this course. However, teamwork had less effect on students' performance compared to hands-on practices and building architecture from the scratch. The reason for this might be that students could possess prior knowledge before the course has begun.

One of the questions in the questionnaire is about future job fields. Results show that students would want to work with "low-level systems" before they took the CA course with a point of 4. However, after they have taken the CA course, their interest increased up to 5 points on average which means a 25% increase in general. Based on this result, it can be assumed that the interest of students in computer hardware/ engineering jobs might be driven not only by their capabilities, but also due to their enjoyment during the course.

Besides the analysis of the questionnaire, students and instructors had a one-hour meeting to discuss results and further opinions. The following items from the discussion are worth noting: a) Implementing the MIPS simulator before the CPU design phase made students confident

[1] MuSe Architecture Verilog
[2] DoMe Architecture Verilog

and aware of what they are doing in the CPU design phase. b) Since the project includes both programming and electronic skills, students preferred to perform their studies as a group rather than work individually. c) Students state that having discussions and presentations in each phase of the project made them better understand the topics, although this approach incurred more workload for them during the studies. d) Although students could not finish the physical implementation part due to COVID-19 conditions, they believe that implementing the designs in an FPGA board instead of using ICs and breadboards would make them learn about embedded systems. Furthermore, since university laboratories have enough FPGA boards for students, the physical implementation part would be cost-free compared to TTL implementation. e) Students claim that they made their

7 DISCUSSION

The CA course has extensive course content such that it is hard to complete all details and fundamentals of the course in a duration of 14 weeks. While the 3-phased processor design project covers the basics and fundamentals of course content, it also transformed the information that could be very abstract in students' minds into a concrete experience thanks to the discussions, peer reviews, presentations, and interviews used during the project. ~~However, the scope of the project does not involve multi-cycle, pipelined architectures.~~ Hence scope can be extended or limited by instructors according to students' capabilities and backgrounds. It would be better if students have intermediate-level programming, logic design, and HDL development skills to achieve the most benefit.

The result of the questionnaire conducted with students after the project is consistent with previous statements and related works. Students mainly point out the importance of simulator and processor design phases. Simulators contributed to students in terms of theoretical knowledge and the processor design phase forced them to question and address the abilities of main units in Von Neumann architecture. HDL implementation gained a different perspective to students but it is important to remark that there is a comprehensive effort of students. However, as students state in the questionnaire, the workload of this approach might bother students and have negative effects on the learning process in some special situations.

Previous studies generally built their research questions upon either designing a better-performance CPU or designing CPUs and tools for education. Many studies are demonstrating, implementing, and evaluating CPU designs from scratch. Furthermore, there are various surveys for different courses including CA to evaluate students' opinions and possible advancements to chronic problems. These surveys shed light on the need for practical experience in theoretical courses However, a CA or related course professor could take this approach, including documents and videos which are publicly available, and use it in their class. Furthermore, lecturers can replicate this approach by adapting it to be used in different fields.

While we tried to fill the gap in improving teaching CA course, the present study lacks a physical implementation phase. The physical implementation of design in the laboratory is one of the biggest and more deductive tasks that give students a chance to evaluate their design performance statistically. Although this was the final task of the project, students could not complete the task due to the Covid-19 pandemic, as project members could not come together. ~~An FPGA implementation as a final assignment to students by considering extra workload.~~

~~The study showed that students involved in hardware design and implementation have a very~~ good understanding of the fundamentals of assembly language, as they simulate the instructions required for assembly language, which improves their ability to understand hardware and software interfaces. This ability would help them to work in building complex computational systems. Furthermore, results state that studying CA course has a significant effect on students' choices in their careers.

8 CONCLUSION & FUTURE WORK

Recent developments in various computer science fields attract students and make them work on more programming and related topics, although in these fields people usually get paid less compared to hardware engineers. Furthermore, several courses that are hard to teach

due to their complexity, may not be ~~internalized~~ by students and ~~this can lead to a situation where students' careers might be affected~~. ~~Therefore~~, this study focused on how to improve students' performances to make sure that they have the utmost benefit from the CA course. In summary, our approach proposes a 3-phased processor design project to be used in CA courses and make it publicly available to let instructors replicate or even enhance it. The approach is a combination of forcing students to creative thinking and hands-on practice by enriching with peer-reviewing and public presentations. It is believed that this approach can be used as a starting point in other computer engineering courses that are theoretical and need low-level computer understanding. This article reveals how students design a processor and how working together in teams increases knowledge diffusion by pointing out that students focus on different aspects of the CPU. The study also demonstrated that once the students are guided carefully, they can design a fully functional processor and its assembly language. The gain of the present study can be summarized as: *"CA course would be very interesting and educatory when proper tools and assignments are provided."*.

Future work ~~of this study~~ will be following tasks to direct scientists and educators: a) the scope of this approach in CA course can be investigated and researchers propose different approaches to cover different contents such as pipelining while keeping students motivated. b) while fulfilling assignments students sometimes face difficulties with tools (Omer et al., 2021). Researchers can change the tools used in this study with tools that are easy to use and record effects on students' development to enrich our, such as RISC-V Online Tutor (Morgan et al., 2021). c) this approach has a high workload for students. The effect of each phase can be investigated, and less effective phases can be discarded. Hence, educators can achieve the same or comparable results with less workload.

ACKNOWLEDGMENTS

The authors would like to thank Ömer Metin, Mustafa Çataltaş, and Sevcan Doğramacı for helping in designing and reporting processes.

A SUPPLEMENTARY

A.1 MuSe Architecture

MuSe Architecture instructions whose format is given in Table 1 are shown in Table 3 with a corresponding explanation. MuSe Architecture designers have an instruction called ***syscall***, which was not mandatory, for enabling programmers to display the contents of registers on the LCD screen when it was called. It does not have an effect on the simulator but it would be useful in physical implementation. ***jr*** and ***syscall*** instructions are exceptions that do not utilize target and destination registers, but they are categorized as R-Format due to their use of source registers

Table 4 shows mathematical operations supported by MuSe Architecture and corresponding control line values. There are eight different operations represented by three different control lines in MuSe Architecture ALU design.

Control signal values for each instruction in MuSe Architecture are provided in Table 5 to help researchers to replicate the MuSe Architecture design.

A.2 DoMe Architecture

Instructions in DoMe architecture are shown in Table 3 with usage in desktop simulators. It remarks that suffix "*-c*" can be used only in R-type instructions. Furthermore, the ALU operation list for DoMe Architecture shown in Table 4 contains more operation and ALU control lines (F3) than the one in MuSe Architecture's ALU Design.

Just like MuSe Architecture, control signal values used in DoMe Architecture are shared in Table 6.

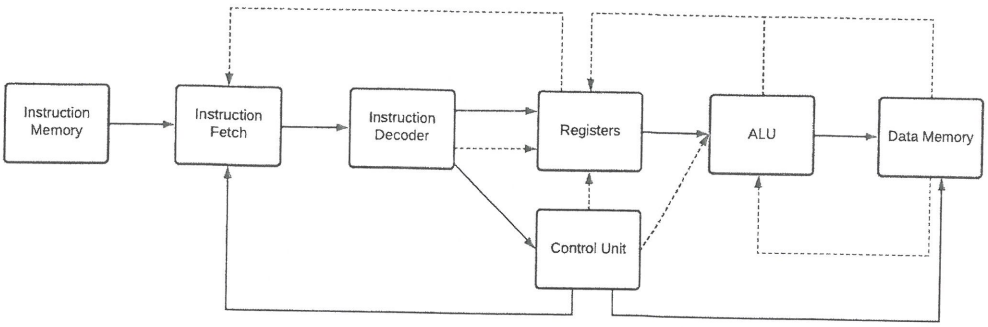

Figure 1. Block-Diagram of a Simple CPU (Wikipedia Contributors, 2022).

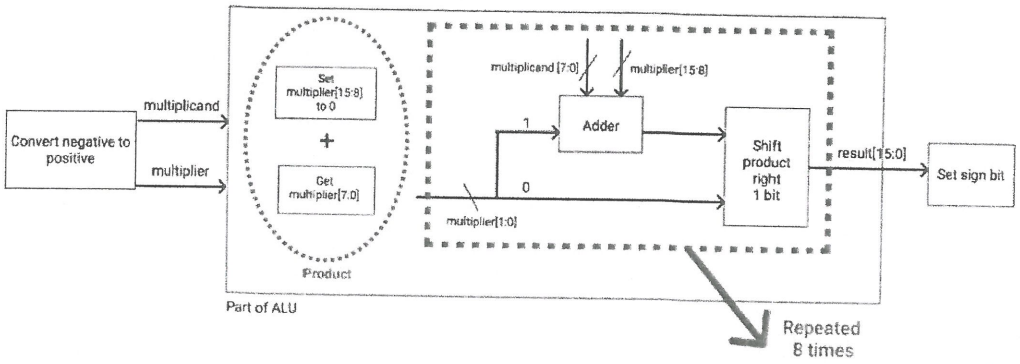

Figure 2. Multiplication Algorithm Design(Patterson and Hennessy, 2016).

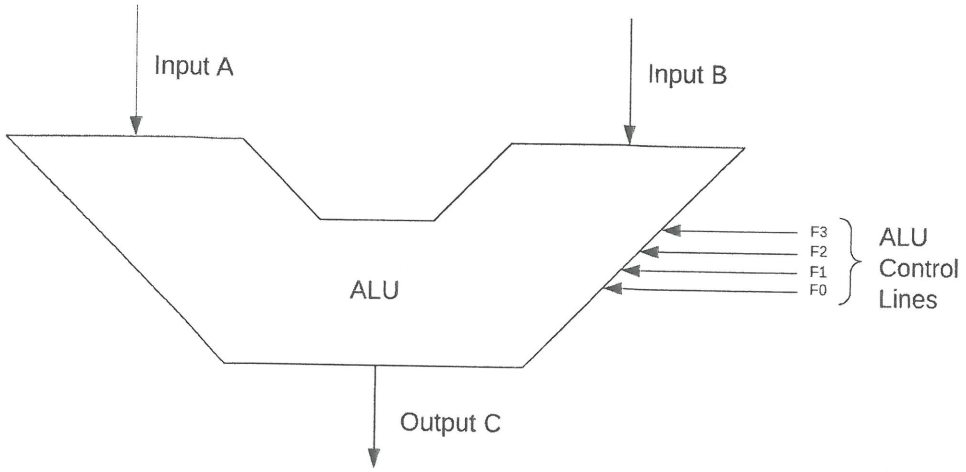

Figure 3. ALU Design.

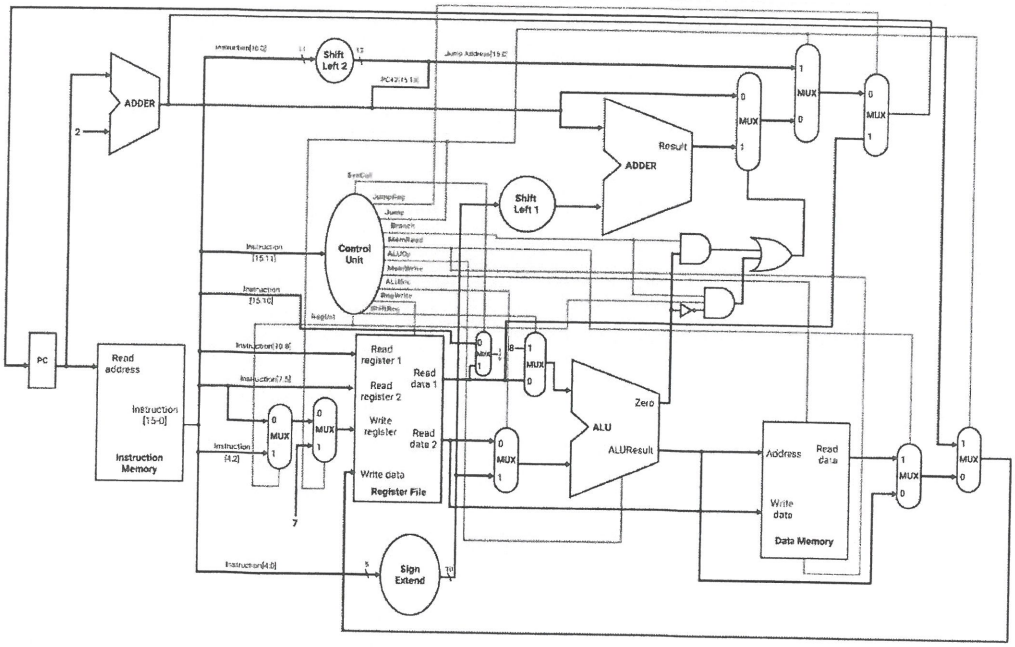

Figure 4. MuSe Architecture Data-Path(Patterson and Hennessy, 2016).

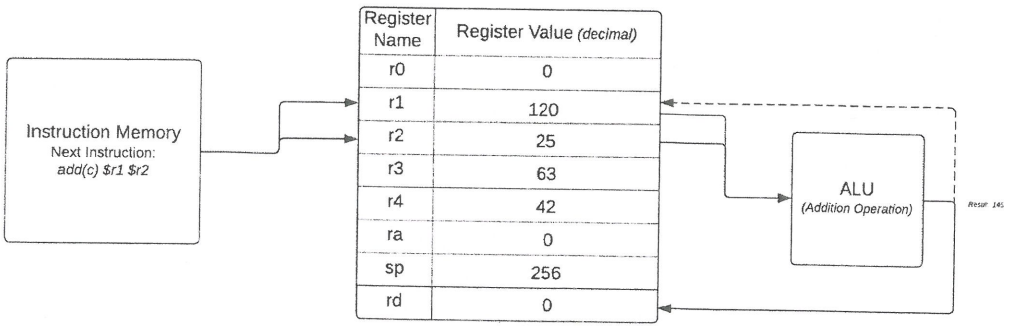

Figure 5. DoMe Architecture Control Bit Example

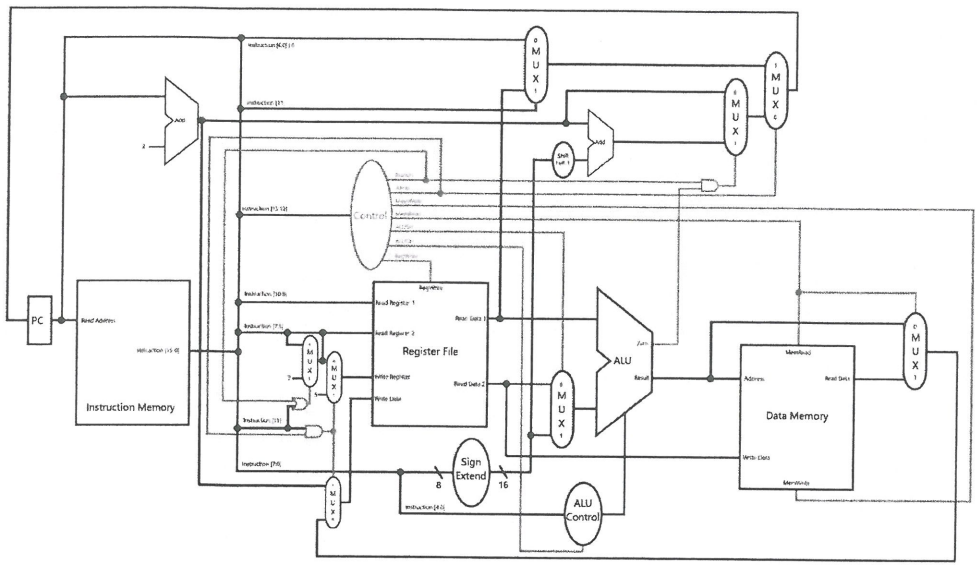

Figure 6. DoMe Architecture Data-Path (Patterson and Hennessy, 2016).

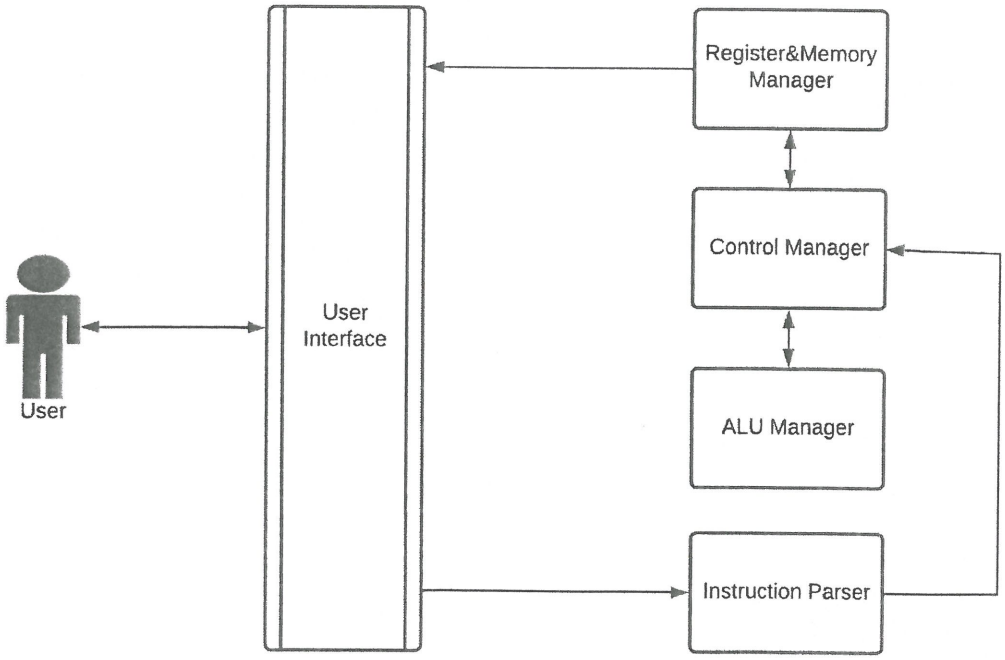

Figure 7. Example Simulator Diagram Used for Teaching CA Course (Wikipedia Contributors, 2022)

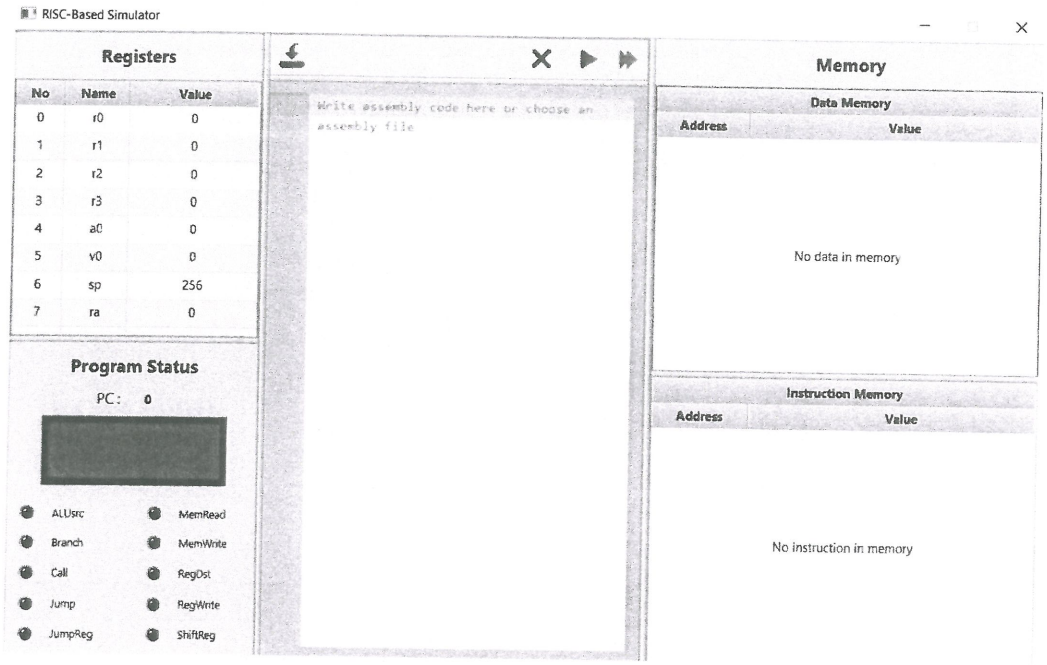

Figure 8. MuSe Architecture Desktop Simulator Screen.

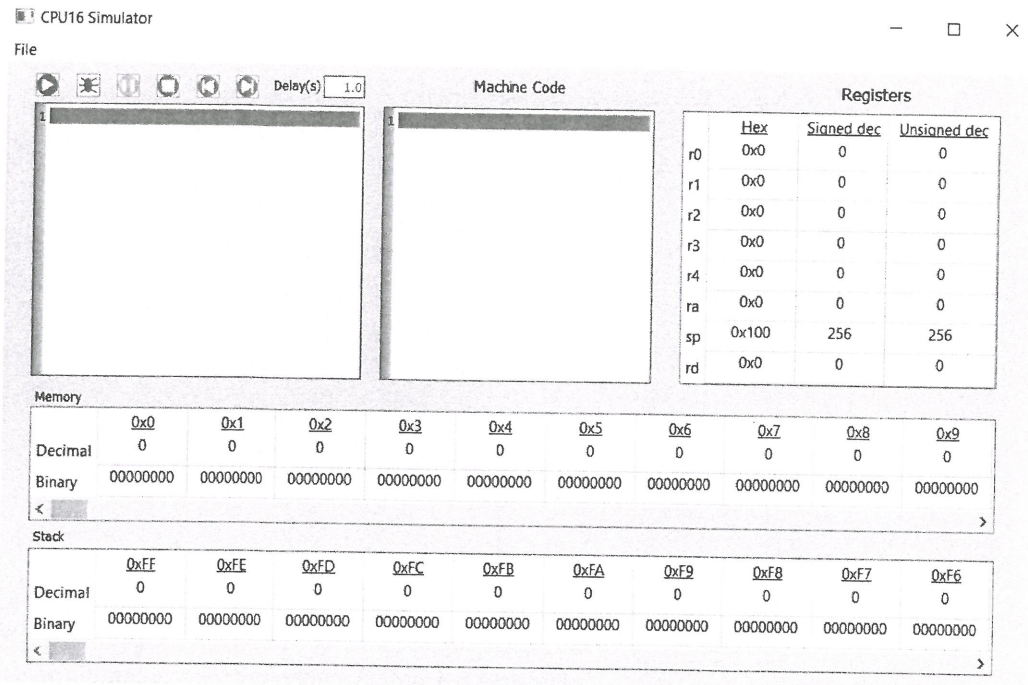

Figure 9. DoMe Architecture Desktop Simulator Screen.

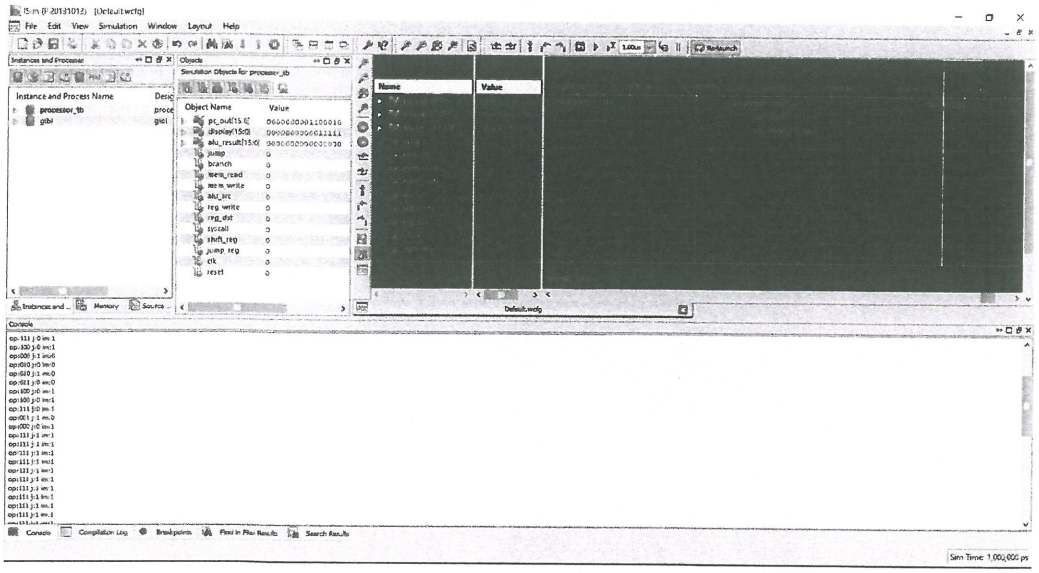

Figure 10. MuSe Architecture Verilog Simulation.

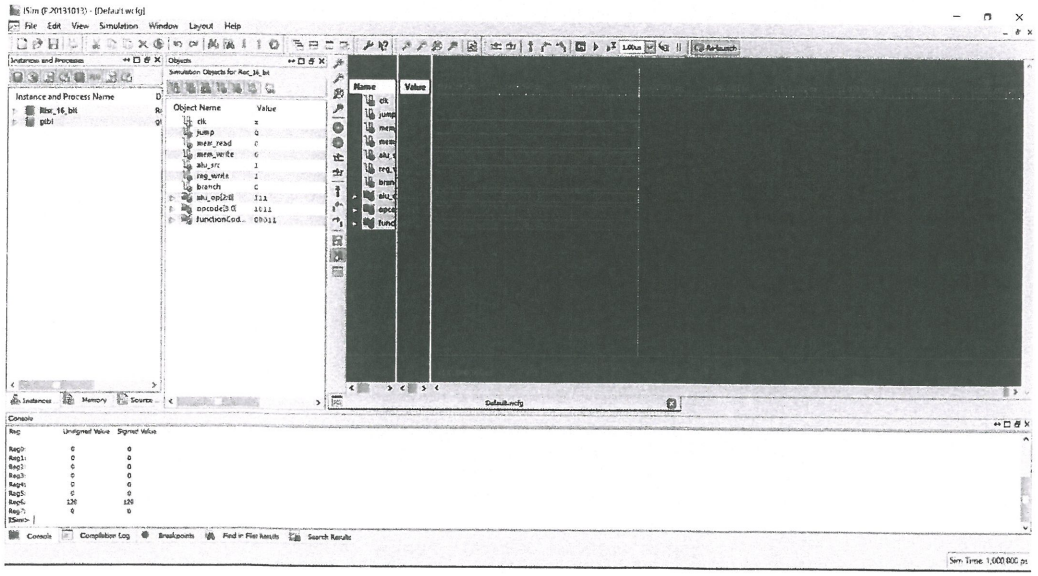

Figure 11. DoMe Architecture Verilog Simulation.

Table 2. Questions and Average Results from Survey

Question	Average Result
Indicate your knowledge and experience level about hardware design before course	4,33
Indicate your knowledge and experience level about hardware design after course	7
Indicate your self-learning ability level before course	6,78
Indicate your self-learning ability level after course	7,71
Indicate how much you wanted to work about low-level systems, such as Computer Hardware Engineering before course	4
Indicate how much you want to work about low-level systems, such as Computer Hardware Engineering after course	5
Indicate your knowledge and experience level about hardware design if you were not exposed to visual tools	6,12
Indicate your knowledge and experience level about hardware design if you were not exposed to group working	6,44
Indicate your knowledge and experience level about hardware design if you were not exposed to any hands-on experiences	4,58
Indicate your knowledge and experience level about hardware design if you did not implement and design own architecture	5,28
Indicate your knowledge and experience level if you have used only simulators and tools shared on the internet	6,14
Indicate your awareness of developing your software by considering hardware abilities and limitations	6,47

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

MuSe Architecture	R-Type	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
		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
		001	1	0	rs	rt	rd	unused	JR
	I-Type	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
		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
	J-Type	000	1	0	imm[10:0]				JAL
		001	1	0	imm[10:0]				J
DoMe Architecture	R-Type	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)
		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)
	I-Type	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
		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	000	0	0	0	0
	sub	1	0	1	0	0	0	001	0	0	0	0
	mul	1	0	1	0	0	0	100	0	0	0	0
	and	1	0	1	0	0	0	010	0	0	0	0
	or	1	0	1	0	0	0	011	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	001	0	0	0	0
	bne	1	0	0	0	0	1	001	0	0	0	0
	sw	0	1	0	0	1	0	000	0	0	0	0
	lw	0	1	1	1	0	0	000	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	0	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	1	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	0	0	1
	bne	0	0	x	0	0	0	0	1	0	1
	lw	1	1	1	0	0	0	0	1	0	1
	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