DOI: 10.1109/TLT.2019.2903714

Wepsim: an online interactive educational simulator integrating microdesign, microprogramming, and assembly language programming

Félix García-Carballeira*, Alejandro Calderón-Mateos*, Saúl Alonso-Monsalve*, Javier Prieto-Cepeda†
,
Computer Science and Engineering Department. University Carlos III of Madrid.
Av. Universidad, 30. 28911 Leganés, Madrid, Spain.
* {*fgcarbal, acaldero, saualons*}*@inf.uc3m.es*
† {*javpriet*}*@pa.uc3m.es*

Abstract—**We have three primary goals in our educational project. First, we want to provide a robust vision of how hardware and software interplay, by integrating the design of an instruction set - through microprogramming - and using this instruction set for assembly programming. Second, we want to offer a versatile tool where the previous tight vision could be tested beyond handwritten exercises. This is called WepSIM and provides an initial elemental processor with a microprogrammed subset of the MIPS instruction set. Besides, WepSIM is flexible enough to be adapted to other instruction sets or hardware components (ARM, x86, etc.). Third, we want to extend the activities of our University lab lectures (fixed hours in a fixed place) so that the students could learn using their mobile device in near any location and any moment of the day.**

This article introduces how WepSIM[1] has improved the teaching of the Computer Structure course by empowering students with a more dynamic and autonomous learning process. In this work, we show the results obtained during the first usage experience - three years - in the Computer Structure course of the Bachelor's Degree in Computer Science and Engineering from the University Carlos III of Madrid.

1. Introduction

There are several interesting simulators used to teach Computer Structure courses. In general, each simulator is used to explain a specific topic of the subject so that students can learn each of the fundamental aspects of the course: assembly, cache, or CPU, separately. Those tools are specific to each aspect of the course; there are used using traditional PCs during fixed lab hours (or event at home).

This diversity does not allow students to have a global view of the system since they do not study the integration of all the elements of the computer and their relationship. Besides, the most realistic simulators are also the most complex. For instance, in our experience, students use to have many problems to understand how interruptions work. The teaching of interruptions is essential for Computer Structure, critical for Operating systems courses, and it is also is a suitable introduction to asynchronous behavior used elsewhere. It is tremendously difficult to get students to understand how the hardware generates an interruption, how the CPU intercepts it, how the CPU searches for the associated handler, how the handler code is executed, and finally how the execution returns to the line of code previously interrupted. There is a continuous interplay between the hardware, the firmware (microcode for example), and the assembly code, which forces the usage of different simulators to explain each part, and then to devote a tremendous effort to link all these concepts. The cost of misunderstanding those ideas could lead to inherent issues [1] with a high impact on the students' learning.

The most popular simulators for educational proposes were created to perform a specific laboratory assignment, and they are available only for traditional PCs (laptops or desktop computers). These conventional PCs usually include a user manual for help, but their goal is not to be a learning material (based on use cases). Moreover, current students spend most of their time using mobile platforms - including smartphones, tablets, Chromebooks, etc. - and they demand interactive and online learning tools - that may be used on a daily basis - rather than the current PC tools.

Given these challenges, our main goal is to provide a modular and straightforward educational and online simulator for Computer Structure courses that can be used by both the students - to learn the topics mentioned above and improve their skills - and the professors - to teach in a better manner and make their work more accessible. The simulator may be used to teach microprogramming and how a simple CPU works, how to use the firmware - through microcode - to create assembly programs, and how the assembly code interacts with both the hardware and the operating system at the same time. We want this simulator to be intuitive and user-friendly, so students do not get lost in irrelevant details, but at the same time, to be very similar to what happens in actual hardware and system software. We also want this simulator to be portable: to be used in smartphones, tablets, but also in desktop computers, and including as much training material as possible.

In this paper, we present WepSIM, an educational and

1. https://wepsim.github.io/, https://wepsim.github.io/wepsim/

online simulator that we have designed and implemented to archive the previously commented goals. WepSIM simulates the circuitry of an elementary processor interactively and allows the user to both define the microcode of the processor and to implement programs in assembly code using the instruction set defined within the microcode. Its circuitry can be modified or extended; it lets students see how circuitry, firmware, and assembly interplay, and it allows students to test paper exercises using both a mobile device or a desktop computer (and test variations of the tasks in an interactive way).

We have also used WepSIM in two Teaching Innovation Projects of our University; in both of them, the students have demonstrated to work autonomously - without the permanent supervision of the professors - and also to verify and validate the designing and resolution of complex problems. The results of providing WepSIM to be used in laboratory classes, and letting students test the exercises they solve by hand in WepSIM are outstanding: the majority of the students are more confident to face the final exam, and they also improve their grades in both the assembly and microprogramming exercise in that final exam. What is more astonishing is that we achieved those improvements even when the ratio 'students per teacher' was increased.

The rest of the document is organized as follows: Section 2 describes the architecture and the hardware model of the simulator. Section 3 introduces the elemental processor that is simulated by using the WepSIM architecture (and the hardware model previously described). This section also describes the microcode and the instruction format. Section 4 describes the main aspects of the implementation process and Section 5 presents the evaluation of the simulator. Section 6 reviews the related work. Finally, Section 7 concludes the paper and presents some future work.

2. The WepSIM Architecture and Hardware Model

Figure 1 shows the architecture of WepSIM. The starting point is the hardware model, which describes the processor to be simulated. It includes the CPU, the main memory, and some I/O devices: keyboard, screen, and a simple I/O device. The hardware model describes the global state of the processor, and it is read by the simulation engine, which updates the global state of the processor for the next clock cycle.

The simulated Control Unit stores the control signals of each cycle in a control memory. These control signals form all the microprograms for every instruction the CPU works with; the CPU always fetchs for retrieving the next instruction from memory and decodes it. The microcode - the content of the control memory - plus the instruction format - instruction parts and its length - is described in a text file. The software model always reads this file and translates it into the corresponding binary code to be loaded into the simulated CPU.

The simulated memory stores the instructions and data values described in a text file with the assembly. The assembly used is the one defined in the microcode format, and the software model translates the assembly into binary code and loads it in the main memory by using the microcode loaded first.

The simulation engine asks the software model subsystem for the microcode, the instruction format description, and the main memory content. The binaries are loaded into the hardware model elements, and then the simulation engine subsystem updates the global state in each clock cycle. The simulation controller subsystem controls when the clock cycle is updated, and when the global state is shown. The simulation UI subsystem updates the user interface and receives the user requests through the user interface events. Upon receiving a user request, the simulation UI subsystem sends the request to the simulation controller. As it can be seen, a very simplified Model-View-Controller (MVC) is used as a base for the WepSIM architecture.

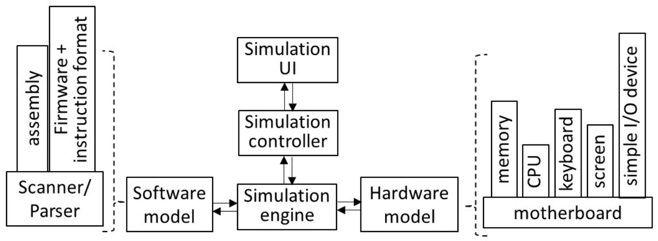


Figure 1: WepSIM architecture.

2.1. Hardware Model

The Figure 2 introduces the model behind the hardware model subsystem. Each element of the circuit could be described as a black box with some possible inputs, some possible outputs, and some control signals to manage the potential transformation of the inputs into the outputs. The hardware model subsystem transforms this black box into two sets of objects: states and signals. A state has an identification (the name), the value (an integer value), and an initial value (the default value). Each value is an integer within the associated range, given by the bits this state uses to be represented. A signal is a particular state that controls the value of other states or signals. Two additional attributes are related to signals (and not to states): the type of signal (level or edge), and the behavior. For each value of the signal, a string describes in a simple language what the signals move or transform. This simple language is composed of instructions that represent elementary operations.

The T4 tristate, for example, has two states: the BUS_IB and the REG_RT1 states. Both represent the value in the internal bus (BUS_IB) and inside the register RT1 (REG_RT1). Additionally, a T4 signal controls when the value inside the register RT1 is sent to the internal bus. This T4 signal is a level signal (type: L) that on a zero value its behavior is not doing anything ("NOP"). But if the T4 signal has a value of 1, the behavior is to copy the value

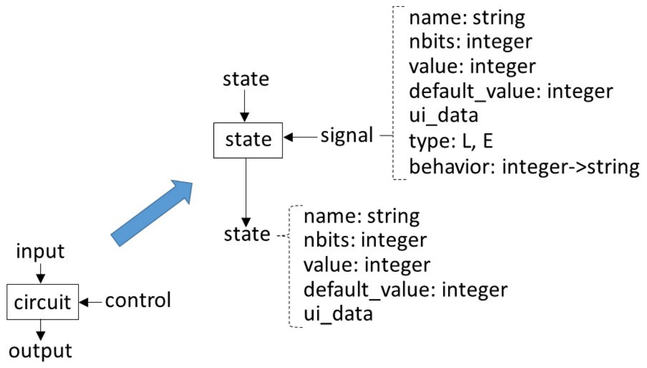


Figure 2: How the hardware is modeled.

in the register RT1 into the internal bus, as it is described by "MV BUS_IB REG_RT1".

For example, register REG_RT1 is similar. It has two states: the content in the register RT1 and the one in the internal bus. The signal C4 controls when the value in the internal bus is loaded into the register RT1. The difference here is the type of signal: C4 is an edge signal (type: E), so at the end of the clock cycle (if C4==1) the behavior is to copy from the bus into the register ("MV REG_RT1 BUS_IB").

3. The WepSIM Elemental Processor

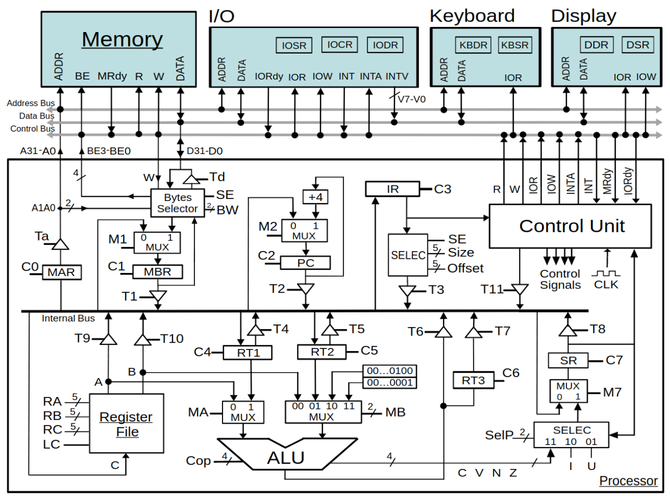


Figure 3: The proposed Elemental Processor.

By using this simple model, it is possible to define all the elements of our elemental processor. Figure 3 introduces the proposed Elemental Processor (named EP hereinafter). The elements in this EP consist of 53 states and 65 signals, in a similar way as in the examples discussed before (in fact, T4 and C4 are included in the 65 signals, and REG_RT1 and BUS_IB are included in the 53 states). The 'behavior language' includes 48 instructions.

The EP has a memory module, a keyboard and a display device, and an I/O generic device used for working with interruptions. Also, a CPU is included, labeled in Figure 3 as Processor. Internally, this CPU has several components at the same time. It has a register file (with 32 registers) and two additional registers - - RT1 and RT2 - not visible for assembly programmers. Values from those registers can be sent into the ALU, that can perform up to 15 arithmetic and logic operations (addition, and, or, etc.). The result can be either stored in a temporary register - RT3 - or sent to the internal data bus. The State Register (SR) can be updated with the flags from the last ALU operation (most common flags are included, such as overflow, negative, zero, etc.). The reader can find that the PC register has its own 'plus four' operator. The IR register has a selector module that lets us extract a portion of the IR binary content, and this part could be sent to the internal data bus. The MAR and MBR registers are used to store the address and the content in this address from/to memory, respectively. The selection circuit lets us indicate the portion of the word from/to memory we want to use. Finally, the Control Unit generates the control signals for each clock cycle.

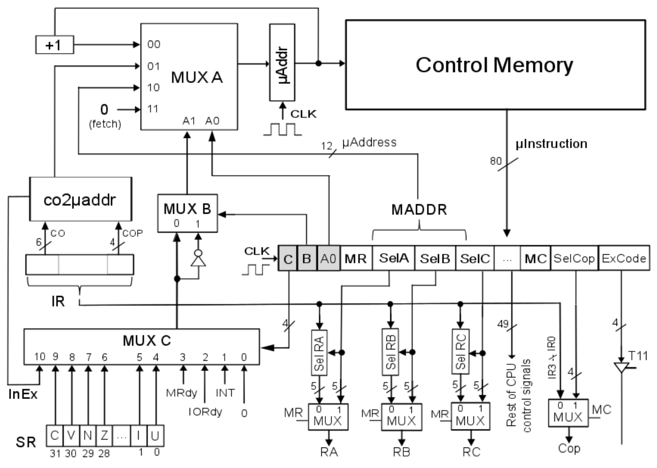


Figure 4: The proposed Elemental Processor: the Control Unit.

Figure 4 shows the Control Unit in detail. The signals for the current clock cycle are stored in the microinstruction register. The content of this register comes from the control memory at the address that the microaddress register points to. This address can be updated with the current address plus one, an address from the microinstruction (overlapped with SelA, SelB and partially SelC that can be conditionally selected with 'MUX C' and 'MUX B'), the first microaddress associated to the operation code field of the instruction in the IR register, and finally the zero value (the reset/fetch microroutine address).

3.1. Microcode

Once the proposed hardware model is defined and imported into the simulator, the next step is to define the microcode that orchestrates it. The instruction format is defined in a text file along with the associated chronogram. The listing 1 shows an example of how the "load immediate" instruction (li) could be defined.

```
li reg val {
   co=000010,
   nwords=1,
   reg=reg(25,21),
   val=inm(15,0),
   { (SE=0, OFFSET=0, SIZE=10000, T3=1,
      LE=1, MR=0, SELE=10101, A0=1, B=1, C=0) }
}
```

Listing 1: Example of instruction format and it associated chronogram (microprogram).

The file with the fetch chronogram and all instruction chronograms defines the microcode for the WepSIM platform. We introduce the possibility to define different instruction sets. We started with a subset of the MIPS instruction list, but instructions from other instruction sets could be defined too.

```
.text
main:  li  $t3 8
       add $t6 $t3 $t3
```

Listing 2: Example of assembly source code, using the instructions described in the microcode.

The 'co' field identifies the instruction code, and it is a 6 bits number. This lets us define up to 32 different instructions. To enhance this number of instructions, 4 bits from the instruction could be used for the ALU selector so the arithmetic and logic instructions can share the same instruction code. Therefore, up to 47 instructions (31 + 16) could be defined. When the WepSIM loads the microcode, each instruction code has associated the start address in the control memory where its chronogram will be stored. This table with two columns, the instruction code, and the associated starting address in the control memory is loaded into the *co*2μ*Addr* ROM that is shown in Figure 4.

For each instruction field (*reg* and *val* in the example of the listing 1) it is defined the initial bit, the ending bit (both included), and the type of field (register, immediate value, absolute address, and relative to PC address).

Once the microcode is loaded into the WepSIM simulator, it is possible to load an assembly file that was built up from instructions defined on the former microcode. In listing 2 is shown an example of the assembly source code. This particular example (listing 2) shows a MIPS-like source code. The load immediate (li) and the add (add) instructions were defined in the microcode previously. The WepSIM simulator can check the syntax errors, and it builds the binary by filling the fields described in the microcode with the corresponding binary information. Figure 5 shows an example of how the "li $2 5" instruction is translated into binary.

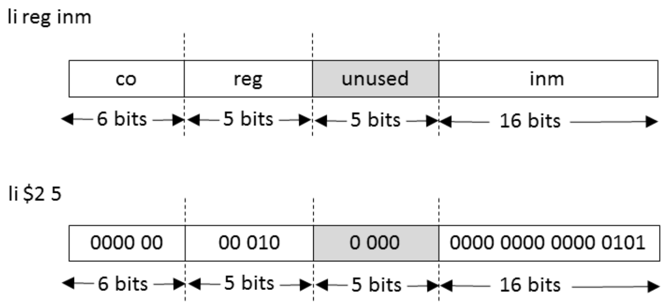


Figure 5: Instruction format described in the microcode and example of its binary translation.

4. Implementation

There are some dependencies among the hardware elements so when some state or signal changes, other signals must be evaluated. In order to deal with the signals-states dependencies, WepSIM tries to reassemble as much as possible what happens in an actual hardware: signals are treated one by one, and when some of them change, then the signals that can be affected are re-evaluated. WepSIM uses a particular behavior for that: "FIRE <signal>". For example, if a new value in the internal bus means that some multiplexer has to check if the new value has to be propagated, we can use "MV BUS_IB REG_RT1; FIRE M7", being M7 the control signal that directs the described multiplexer. WepSIM detects loops while processing dependencies - by using a stack of dependencies being analyzed - and avoid the infinite loop.

4.1. Cross-Platform

The prototype introduced in this paper is implemented in HTML5, so all platforms with an Internet browser (smartphones, tablets, laptops, and desktop computers) can execute it. WepSIM works thanks to Mozilla Firefox (version 50+), Google Chrome (version 55+), Microsoft Internet Edge (version 38+) browsers, or Apple Safari (version 11+).

In order to execute WepSIM even without an Internet connection, we pack it as a mobile application thanks to two options: by using the Apache Cordova project, and more recently thanks to the Progressive Web Apps initiative. Both options let us to deploy WepSIM for Android, iOS, Windows, MacOS, etc.

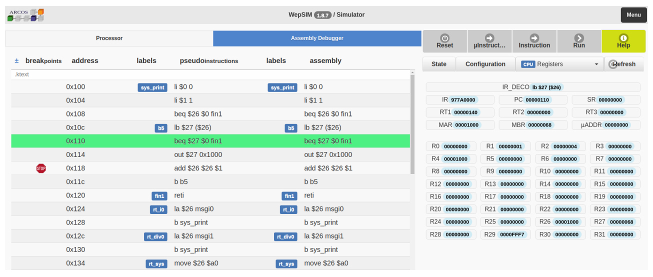
The WepSIM source code is composed (without external dependences) of 24 files and it is available at https://github.com/acaldero/wepsim. It consists of $\sim 10,000$ lines of JavaScript code and $\sim 1,700$ additional lines of HTML plus CSS. This source code can be compressed and minimized in about 200.000 bytes in total (around 195 KiB). It only requires the well-known frameworks/libraries: JQuery, JQueryUI, JQuery Mobile, KnockOut and BootStrap. It also uses the glyphicons icon set (thanks to Jan Kovarik for providing us this free version).

4.2. The ‘Test and Learn’ User Interface

The user interface has three views: simulation, microprogramming view, and assembler view. The three views are interconnected to allow students to microprogram an instruction set (MIPS, ARM, Z80, etc.), then program an assembly application with the defined instruction set, and finally execute the assembly application.

Each one of the 3 view provides as much feed-back as possible to students so they can test and learn. For instance, the circuit cables change its color when they are used, control signal change its color when they are activated, and so on. Moreover, each view has a help entry with the associate explanation, there are 12 examples by default, and there are 2 initial tutorials that quick cover how the user interface is used.

Figure 6 shows an example of the assembly debugger execution, and Figure 7 shows the processor simulator where by clicking on signal name open a popup where students access to the associated help, and the form to change the current value of this signal.



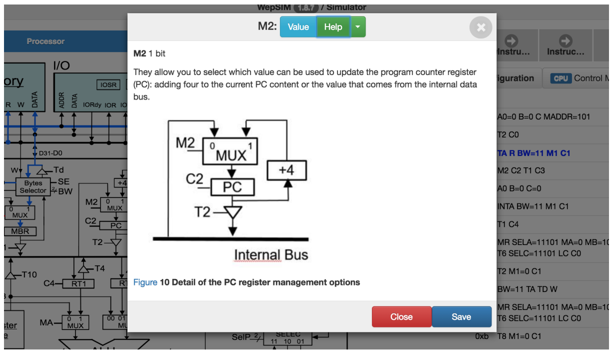
breakpoints	address	labels	pseudoinstructions	labels	assembly
.ktext					
	0x100	sys_print	li $0 0	sys_print	li $0 0
	0x104		li $1 1		li $1 1
	0x108		beq $26 $0 fin1		beq $26 $0 fin1
	0x10c	b5	lb $27 ($26)	b5	lb $27 ($26)
	0x110		beq $27 $0 fin1		beq $27 $0 fin1
	0x114		out $27 0x1000		out $27 0x1000
STOP	0x118		add $26 $26 $1		add $26 $26 $1
	0x11c		b b5		b b5
	0x120	fin1	reti	fin1	reti
	0x124	rt_i0	la $26 msgi0	rt_i0	la $26 msgi0
	0x128		b sys_print		b sys_print
	0x12c	rt_div0	la $26 msgi1	rt_div0	la $26 msgi1
	0x130		b sys_print		b sys_print
	0x134	rt_sys	move $26 $a0	rt_sys	move $26 $a0

IR_DECO lb $27 ($26)

IR 977A0000	PC 00000110	SR 00000000
RT1 00000140	RT2 00000000	RT3 00000000
MAR 00001000	MBR 00000068	μADDR 00000000

R0 00000000	R1 00000001	R2 00000004	R3 00000000
R4 00001000	R5 00000000	R6 00000000	R7 00000000
R8 00000000	R9 00000000	R10 00000000	R11 00000000
R12 00000000	R13 00000000	R14 00000000	R15 00000000
R16 00000000	R17 00000000	R18 00000000	R19 00000000
R20 00000000	R21 00000000	R22 00000000	R23 00000000
R24 00000000	R25 00000000	R26 00001000	R27 00000068
R28 00000000	R29 0000FFF7	R30 00000000	R31 00000000

Figure 6: Main view of the assembly debugger.



M2 1 bit

They allow you to select which value can be used to update the program counter register (PC): adding four to the current PC content or the value that comes from the internal data bus.

Figure **10 Detail of the PC register management options**

Figure 7: Help details of the ‘M2’ signal.

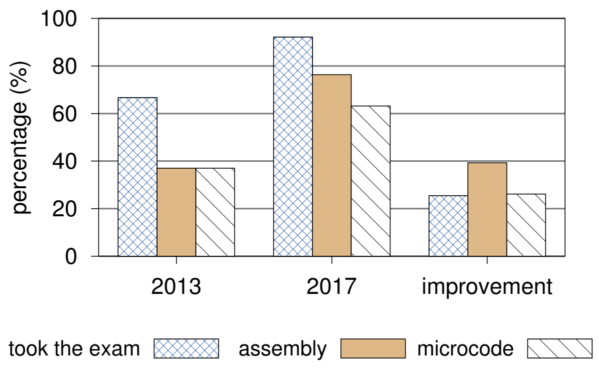
5. Evaluation

An important complement to perform exercises by hand is to provide a way to make these activities to come interactive, so students can make changes and study their effect. WepSIM can help to increase the curiosity of the students all times, but especially on the first exercises where WepSIM helps to learn how an elemental processor works. Helping on the initial exercises empower students to continue with the exercises proposed in the literature in a better way.

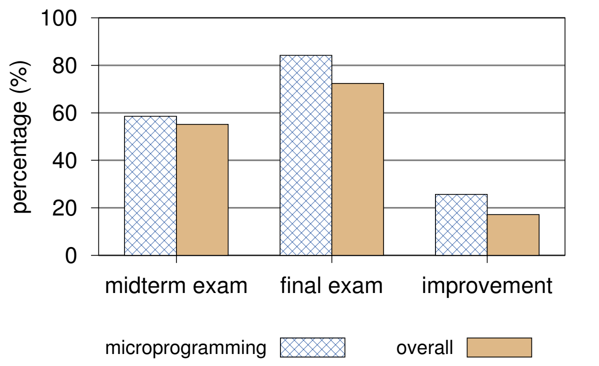
A critical aspect of this first contact with exercises is to let students make experiments at any moment, not only in the hours of the day associated with the laboratory work. WepSIM mobility and portability are essential to provide a better adjustment of the learning process of the students. And because WepSIM includes help material and examples, it can be used as a more autonomous learning tool than the existing ones (as far as authors know). The following subsection describes the evaluation of the first WepSIM experience on that.

5.1. Grade Improvements

We have evaluated the impact of using WepSIM in three teaching groups of near 100 students in total, and the results are quite similar in all groups. To be more near to the students’ point of view, we are going to show the results for one teaching group in two different courses (Figure 8).



(a) Obtained portion of total grades in the final exams’ exercises of assembly and microprogramming (2013 vs 2017).



(b) Grades comparison between midterm and final exams (both the microprogramming exercise and the overall grade, 2017).

Figure 8: Comparison of student grades (from 27 students in 2013 to 38 students in 2017).

As it is shown in Figure 8a, in 2013/2014 the WepSIM teaching utility did not exist, and from 27 total students in the final exam, only 18 performed the exercises of the exam. In 2017/2018 our proposed WepSIM utility was used (not only as lab supporting tool but as a teaching assistant tool too). From 41 total students in the final exam, 38 performed the exam's exercises. By using WepSIM we were able to empower students to be more confident in its abilities, and the percentage of student that performed the exam increase from 66% (18/27) up to 92% (38/41).

On the other hand, Figure 8b shows the grade comparison between the midterm and the final exam, taking into account both the microprogramming exercise and the overall grade of the 2017/2018 course. As it can be seen, the same group of students greatly improved their grade from the midterm exam - without using WepSIM - to the final exam - after two months of using WepSIM. This Figure demonstrates that there is not only a significant improvement among different courses, but also the same group of students is able to improve their grades after a WepSIM-based learning.

When the number of students per group increase (from 27 up to 41) with the same amount of teachers, then the time available per student decreased. Thanks to WepSIM, the results show that we can improve the learning experience, and the grades of the students, even with high 'student per teacher' ratio.

Only a teaching tool like WepSIM let teachers introduce the elemental concepts and behaviors, then the students could review and extend their abilities and knowledge, and finally, the teachers could solve the students' doubts (and perform some more advanced exercises). Not only more learning tasks are transferred to the student, but also it is done in a more fun and active way. WepSIM becomes not only a lab tool, but also a learning tool where students could test different scenarios.

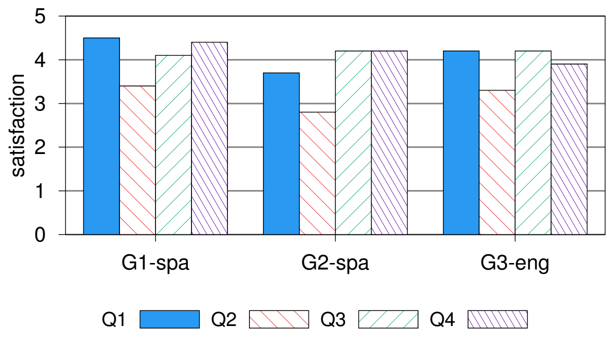


Figure 9: Poll average results from the students.

5.2. Survey Results

After the microprogramming laboratory based on the WepSIM simulator, we performed a simple survey to the students to know some of the learning results achieved with our proposal. The questions of this poll were: (Q1) I agree to participate in new teaching proposals. (Q2) I frequently do hand-made exercises. (Q3) It is better simulator-based exercises rather than hand-made exercises. (Q4) The simulator provides a better understanding of how a processor works.

For each question, the student could rate from 0 up to 5, where 0 means not to agree at all and 5 means totally agree.

The Figure 9 shows the average results from three student groups of forty students each one. The results confirm that we have accomplished our goal.

5.3. Platforms Where the Simulator Was Used

The first study from the Web Server logs analyzed the proportions of computer clients requests that came from the UC3M University or from outside the University. During the last week for the laboratory dead-line, the WepSIM simulator has been used a 7% with IP address from the University,

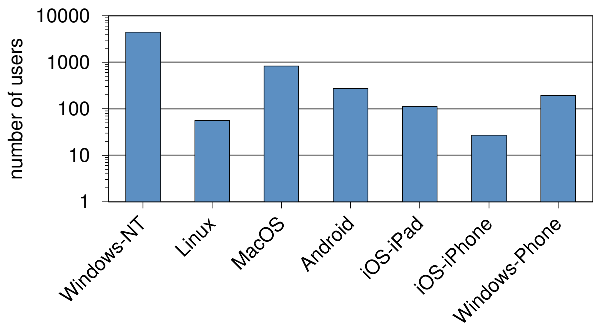


Figure 10: Client operating systems that use WepSIM.

The second study analyzed the previously described logs to determine the Operating System used by computer clients (as they identify themselves). We wanted to analyze whether WepSIM helps students to use any platform of their election. Results are shown in Figure 10, where the y-axis represents the number of access and the x-axis the platform from where the access came. The y-axis uses a logarithmic scale. The x-axis group on the left the desktop-related operating system (Linux, Windows-NT, and MacOS), and on the right group mobile-related platforms (Android, iOS, and Windows Phone). Results show that students were able to use different platforms, we achieved the goal we planned.

5.4. Time Instants When WepSIM Was Requested

We have analyzed the Web service logs in order to study the time instants where the WepSIM has been requested in 2013. We have focused on the last week before the deadline for students to finish the work with the WepSIM simulator. Figure 11 shows the results, where the X-axis is the hour during the day, and the circles over the X-axis represent the access at this moment. As can be seen, there is access beyond the one 100 minutes laboratory class. These accesses are concentrated from nine o'clock to two o'clock in the afternoon, and from 15:00 to 23:00. In summary, these results show that we have facilitated the usage of WepSIM at any time of day, and from different platforms. For 2017 we used Google Analytics in order to know the platform used, and the hours and days where WepSIM has been used. Figure 12 shows the hours and days of the week. The aggregated result are very similar to the ones from 2013.

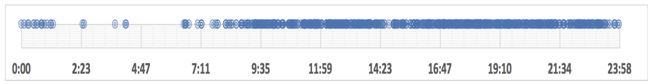


Figure 11: Moments of the day when WepSIM was used in 2013.

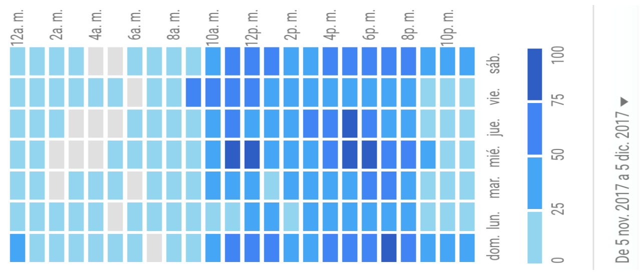


Figure 12: Days and hours when WepSIM has been requested in 2017.

6. Related Work

There are several useful tools for Computer Structure teaching activities although we have not found a tool (like the one introduced in this paper) that: (a) Providing an inter-related vision of the microcode and the associated assembly programming. (b) It allows to define a broad set of machine instructions, and it is based on a hardware model that can be extended or modified. (c) Giving students more flexibility on the platform used (e.g.: mobile devices), and providing student a learning tool as much self-contained as possible (it includes associated online help and context-aware help).

Djordjevic et al. [2] propose a non-intuitive Java Application: it is not easy to see the relationship between the microprogramming and the assembly programming level. It did not include online help, examples, etc. and It is not ready for mobile devices (we did not find the open repository where the project is published).

There are two well-known 32-bits MIPS processor simulators for teaching assembly: SPIM [3], and Mars [4]. Unfortunately, both of them have not an integrated vision for microcode and assembly, they are not designed for mobile devices in mind (even the existing un-official version seeks to resemble the computer version), and is limited to the 32-bits MIPS instruction set.

Other two important simulators we want to highlight are the P8080E [5], and the pc88110 [6]. P8080E was designed for teaching assignments in microprogramming, but it was not intended to be used to teach both microcode and assembler with the same tool, it has not a portable and interactive graphical interface, and the microcode does not include the instruction format so the microprogramming is done by hand in binary. Although PC88110 simulator integrates the effect of a superscalar processor, microcode is not integrated, it focuses on a specific architecture (MC88110), and does not provide support for mobility. Behind this simulator is a fascinating work described in [7] that includes the methodology behind the simulator pc88110 to be used in practical laboratories.

Outside the educational system there are simulators/emulators like OOVPsim [8] or GXemul [9] for working with different architectures such as ARM, MIPS, PowerPC, etc. OVPsim has been used for research parallel computing platform [10], hardware/software co-design [11], etc. Although it is possible to be used for teaching assignments, its level of detail makes not the best tool to start learning.

7. Conclusion and Future Work

This article has introduced WepSIM, a new intuitive, portable, online, and extensible educational simulator. WepSIM is based on a simple yet powerful model that tries to mimic how the hardware elements work. It also provides an integrated learning experience on microprogramming and assembly programming because it is possible to define different instruction sets and to execute source code based on the defined instruction set.

WepSIM allows the students to understand how a processor works smoothly. It can be used on a smartphone, tablet, laptop or desktop computer with a modern Internet browser. In this way, the students can interact with the simulator and learn how the typical hardware elements work, and the mechanism for communicating with the system software.

There are several impressive future works, and we are already working on them at the moment: (a) We plan to integrate a testing module into WepSIM to check one microcode with several assembly programs and vice-versa. (b) A tiny footprint operating system written in assembly is planned too, so students could learn better how the operating system interacts with the hardware. (c) More pieces of hardware can be added, such as the ones to support single-precision floating point arithmetic. (d) We are studying to integrate new devices such as a sound card - with DMA - or a cache memory.

References

[1] S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Moritz lipp1, michael schwarz, daniel gruss, thomas prescher 2, werner haas 2," *arXiv preprint, Computer Science, Cryptography and Security*, 2018.

[2] J. Djordjevic, B. Nikolic, and A. Milenkovic, "Flexible web-based educational system for teaching computer architecture and organization," *IEEE Transactions on Education*, vol. 48, no. 2, pp. 264–273, 2005.

[3] J. R. Larus, "SPIM," Feb. 2016. [Online]. Available: http://spimsimulator.sourceforge.net/

[4] P. Sanderson and K. Vollmar, "MARS," Feb. 2016. [Online]. Available: http://courses.missouristate.edu/kenvollmar/mars/

[5] DATSI.FI.UPM.ES, "P8080E," Apr. 2016. [Online]. Available: http://www.datsi.fi.upm.es/docencia/Estructura/U_Control/

[6] M. I. Garca, S. Rodrguez, A. Prez, and A. G. Dopico, "p88110: A graphical simulator for computer architecture and organization courses." *IEEE Trans. Education*, vol. 52, no. 2, pp. 248–256, 2009.

[7] A. G. Dopico, S. R. de la Fuente, and F. J. R. Garca, "Automatización de prácticas en entornos masificados," in *Actas de las IX Jornadas de Enseanza universitaria de la Informática*, ser. Jenui 2003. Spain: Thomson-Paraninfo, 2003, pp. 119–126.

[8] I. Software, "Open Virtual Platforms simulator," Mar. 2016. [Online]. Available: http://www.ovpworld.org/

[9] A. Gavare, "GXemul," Mar. 2016. [Online]. Available: http://gxemul.sourceforge.net/

[10] C. Pinto, S. Raghav, A. Marongiu, M. Ruggiero, D. Atienza, and L. Benini, "Gpgpu-accelerated parallel and fast simulation of thousand-core platforms." in *The 11th International Symposium on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2011, pp. 53–62.

[11] I. Nita, V. Lazarescu, and R. Constantinescu, "A new Hw/Sw co-design method for multiprocessor system on chip applications." in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and system Synthesis*. IEEE C.S., 2009, pp. 1–4.

[12] J. Matak, "Assembly Emulator," Feb. 2016. [Online]. Available: https://play.google.com/store/apps/details?id=gr.ntua.ece.assembly.emulator

[13] F. G. Carballeira, J. C. Pérez, J. D. G. Sánchez, and D. E. Singh, *Problemas resueltos de estructura de computadores, segunda edición*. Ediciones Paraninfo, 2015, vol. 1, pp. 1–307.

[14] J. M. Pérez Villadeamigo, S. R. de la Fuente, R. M. Cavanillas, and M. I. García Clemente, "The em88110: Emulating a superscalar processor," *SIGCSE Bull.*, vol. 29, no. 4, pp. 45–50, Dec. 1997.

[15] S. S. et al, "From NAND to Tetris," Nov. 2016. [Online]. Available: http://www.nand2tetris.org/

[16] J. N. et al, "The Megaprocessor," Nov. 2016. [Online]. Available: http://www.megaprocessor.com/