Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

QtARMSim: an ARM simulator for teaching Computer Architecture

Sergio Barrachina* | Germán Fabregat | Juan Carlos Fernández | Germán León

[1]Department of Computer Engineering and Science, Jaume I University, Castellón, Spain

Correspondence
*Sergio Barrachina. Email: barrachi@uji.es

Present Address
Department of Computer Engineering and Science. Jaume I University. Avda. Vicente Sos Baynat, s/n, 12071, Castellón de la Plana, Spain

Summary

Many of the teaching objectives of Computer Architecture introductory courses are focused on the vision that the programmer of assembly language has of a computer. As a general rule, in order to define these objectives, a specific computer architecture is used which normally conforms to two criteria: being as simple as possible, and at the same time, motivating for the students.
The ARM architecture is an ideal base for teaching Computer Architecture. On one hand, as it is based on the Reduced Instruction Set Computer (RISC) architecture, it is relatively simple. On the other, it is a modern architecture widely used, especially in mobile phones, smartphones and tablets, which is motivating for the students.
To do practical work based on ARM, either an ARM simulator or a development tool on an ARM machine should be used. When designing a course for a first year undergraduate level, we believe that the selected tool should be: easy to use, multiplatform, free, and open.
After evaluating existing options, we decided to develop and release an ARM simulator, ARMSim, and a graphic interface for that simulator, QtARMSim, pursuing the previously stated objectives. This software has already been used in 2014 and 2015 courses. The feedback received from both students and laboratory teachers has been very positive.

KEYWORDS:
Computer Assisted Education, Computer Architecture, ARM Thumb, Simulator, Assembly language

1 | MOTIVATION

Historically, the content of courses on Computer Architecture has followed the frenetic pace set first by the technological advances in the design of large computers and from the 1980s on, by the evolution in the design of microprocessors. Therefore, it can be said that the teaching of Computer Architecture has gone through six main stages: mainframes, minicomputers, the first microprocessors, microprocessors, RISC and post RISC. Each time universities have been able to get access to specific hardware at a reasonable price, this has been widely used as reference material. The PDP-11, the 68000 of Motorola, the 80x86 of Intel, the MIPS and the SPARC are some of the machines and microprocessors which have enjoyed most popularity in the teaching of Computer Architecture. On some occasions, however, hypothetical computers dedicated exclusively to the teaching of architectural concepts have been used[1].

The Computer Architecture courses at the University of Jaume I, Spain, have also passed through the various stages mentioned above; the first system to be used was the 68000 Motorola, later a hypothetical computer was briefly used, and subsequently this was replaced by the MIPS architecture, which was used until the 2013 course.

Early on 2013 the teachers involved in the teaching of these courses proposed decided to change from MIPS to ARM for the following reasons. On one hand, the ARM architecture has many characteristics that distinguish it from other contemporary architectures while at the same time, since it is based on RISC, it is relatively simple[2]. On the other, ARM is currently widely used, especially in mobile phones, smartphones and tablets, which is especially motivating for the students[3]. In fact, over the past two decades, the ARM architecture has exploded in popularity because of its efficiency and rich ecosystem. More than 50 billion ARM processors have been shipped, and more than 75% of humans on the planet use products with ARM processors[4].

Once the decision to make this change was made, the task of redefining all the instruction materials, was also undertaken. It had to be decided which new material should be used for both theory and laboratory on each of the courses that address the Computer Architecture subject. In the case of the Computers Organization course, which is taught on the first course of the Computer Engineering and Computational Mathematics grades of the Jaume I University (Spain), one of the primary objectives was to select an ARM simulator or framework which could be used on the laboratory sessions. This decision hat to be conditioned by the fact that other courses were going to use the Arduino Due platform[5], which is equipped with a microprocessor based on the ARM Thumb 2 architecture.

In previous years, in the laboratory sessions of this course, a MIPS simulator, `xspim`, was used (`xspim` is a previous version of the current `QtSpim`[1] simulator)[6]. The `xspim` simulator was a suitable platform for the practical work in the Computer Organization course for the following reasons: i) it permitted the step by step simulation of the execution of assembly programs, ii) it visualized the content of registers and the memory in a simple way, and iii) it was a free, open and multiplatform software. Given that `xspim` was sufficiently simple, the students were able to concentrate on the development and simulation of programs, rather than on operating the simulator. At the same time, since it was a free multiplatform software, the students were able to download it and practice in their own time and on their own computers.

Although there are quite simulators suitable for teaching courses in computer architecture and organization (see[7] survey for example), we were interested on ARM simulators/frameworks who focused on the fundamentals of computer architecture knowledge unit defined at the Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering[8].

The first option considered for substituting `xspim` for an ARM framework, was the µVision[2] integrated development framework, from the Keil company. This framework is the most widely used in the programming of microcontrollers based on ARM (the Keil software company was bought by ARM in 2005) and is given as an example in books about ARM such as[9] and[10]. It includes C and C++ compilers, assembler, linker, filter, debugger and simulators of various microcontrollers based on ARM.

Adopting µVision would have provided a realistic framework for embedded systems programming. However, its operation would have been slightly more complex than was really wanted for a first year undergraduate course in which the intention was to explain the fundamentals of Computer Architecture rather than programming embedded systems. At the same time, there is only a Microsoft Windows version, it is not open software, and while there is a free of charge version, there are certain limitations: 32 KiB of code (which wouldn't have caused any problem), and the need to register for downloading it (which would require the students to give their personal information in order to be able to use it).

Related to the µVision IDE, the acquisition of a product called Lab-in-a-Box[3] from ARM University, was also considered. This product includes: i) licences of Keil MDK-ARM Professional Software Tool (which forms part of µVision IDE); ii) Freescale cards based on the ARM Cortex-M0+ processor; and iii) teaching material for theory and laboratory sessions. Once we confirmed that the course was based exclusively around the use of C for programming embedded systems, it had to be discarded as not being a valid option for this course.

The next alternative considered was the use of Eclipse DS-5. Just as µVision is related to embedded systems, Eclipse DS-5 is the development framework provided by ARM for ARM System-on-Chip processors. It allows the programmer to use C/C++ and assembly, as well as to debug the code which is being executed in the computer for which it is being developed. Although the usual version of Eclipse DS-5 is neither open or free, there is a version called Community Edition which is. What is more, since it is actually an Eclipse plugin, it is a multiplatform solution.

However, one of the first weak points that was detected was that it does not provide a true simulation environment. Nevertheless, it is possible to use the debugging options to mimic a simulation environment. All that needs to be done is to create a

[1]QtSpim can be downloaded from: http://spimsimulator.sourceforge.net/
[2]µVision Getting Started book: http://www.keil.com/product/brochures/uv4.pdf
[3]ARM Course/Lab: http://arm.com/support/university/educators/embedded/

virtual ARM machine (e.g. with QEMU), install a GNU/Linux operating system on the virtual machine and configure it so that a GDB server can be accessed from Eclipse DS-5 through SSH. In this way, the debugging capacity of Eclipse DS-5 serves to permit the observation of the execution of a program without the need to use specific hardware. It is also possible to follow the same procedure, but using actual hardware, such as a Raspberry Pi. Although we were able to debug code using both methods, with a virtual machine and a Raspberry Pi, both its setup and the use of the Eclipse debugger as a simulation environment was excessively complex for a first year course.

Lastly, the ARMSim#[11] simulator was evaluated. This application allows the simulation of the execution of assembly programs in a system based on the ARM7 processor. It is written in C#, works in Microsoft Windows and, in theory, with Mac OS X and GNU/Linux using the Mono implementation of the .NET environment. This simulator was close to the one that was being looked for this course, and contained interesting characteristics such as the simulation and visualization of peripherals, and the possibility of developing plugins to extend it with new I/O devices. However, it does not currently support the Thumb variant of ARM, which was necessary in order to link the simulator based practical classes in this course, with subsequent practical work in the same course and in others in the same degree, in which the student programs an Arduino Due card (that uses an ARM Thumb2 processor).

So, given that none of the systems that were considered covered all the expected needs for the course, we decided to develop our own ARM simulator[4], with the intention of releasing it so that it could also be used on other Universities introductory courses on Computer Architecture.

The developed simulator consists of a simulation engine, ARMSim, and a graphic interface, QtARMSim. Given that both applications are completely disengaged, anyone interested only in the simulator could develop his own graphic interface, or in the opposite case, the graphic interface could be slightly modified in order to connect it to a different simulator. In addition, a textbook[5] in Spanish has been written, in which the simulator is described in detail and a series of practical introductory exercises are proposed on the ARM Thumb architecture.

An early version of the ARMSim and QtARMSim simulator was described in[12] (in Spanish). Later, in[13] (also in Spanish), the simulation engine was described in more detail, some improvements on the graphic interface were presented, and the feedback obtained from the students and their academic progress before and after using the simulator was presented. This article presents the current version of the simulator, which among other improvements and enhancements, provides: i) a color trace of the instructions that have been simulated, ii) a memory dump at a byte level, iii) a basic LCD display device simulation, and iv) a bundled GCC assembler. These enhancements are aimed at providing the student a better understanding of the execution flow and the memory organization, and an easier installation of the simulator.

The rest of the article is organized as follows. The second section describes the simulator objective. Section 3 describes ARMSim, the simulation engine. Section 4 describes QtARMSim, the graphic interface of the simulator. Section 5 presents the obtained results. And finally, on Section 6, conclusions and future work are given.

2 | OBJECTIVE

The main objective that has guided the development of the simulator has been to provide the students with an ARM Thumb simulator which would allow them to easily acquire the necessary knowledge about the fundamentals of Computer Architecture. In order to achieve this teaching objective, the simulator had to be as simple as possible, free, multiplatform, open, and cover the entire process of development, assemble, and simulation of ARM Thumb assembly code. In addition to these, we considered that it would be convenient for the simulation engine to be disengaged from the simulator graphic interface.

Forcing students to spend longer time understanding poorly usable interfaces than understanding learning content disturbs accommodation of new concepts and overall retention of what is being learned. The user should be involved in the learning process without being overwhelmed[14]. Therefore, the simulator should be as easy to use as possible so that the students could concentrate on the development and simulation of assembly programs, rather than on the operation and configuration of the simulator.

It was also necessary for the simulator to be free and multiplatform, so that the students could have the chance to install it without cost on their own computers, independently of whichever operating system they use.

[4]QtARMSim can be downloaded from: http://lorca.act.uji.es/projects/qtarmsim/
[5]The textbook can be downloaded from http://lorca.act.uji.es/libro/introARM/

Releasing it as open software will allow the students to inspect the code, not only of the simulation engine, but also of the graphic interface. Also, third parties could be involved in its development, or adapt it for their own needs.

At the same time, the simulator should cover the whole process —code edition, assembly, and simulation—, so that the overall student experience could be as simple as possible. This way, the need for the student to alternate between an editor —to write the assembly code—, and a simulator —for assembling and simulating the code—, as happens on other simulation environments, should be avoided.

The last of the features we had on mind when developing this project was that the simulation engine and the graphic interface were as disengaged as possible. This would allow in the future the simulator to be used, even remotely, from different graphic interfaces. Or to use the same graphic interface with different simulation engines. Due to this characteristic, the simulator really consists of two applications: ARMSim, a simulation engine for the ARM Thumb architecture, described on next section, and QtARMSim, a graphic interface for the simulator, described on Section 4.

3 | ARMSIM

ARMSim is a simulation engine for the ARM Thumb architecture. It provides a model of the processor —which includes registers and state indicators— and a memory model —both RAM and ROM—. The model of the processor is able to decode and execute the whole set of instructions from the ARM Thumb architecture. The memory model makes it possible to set the amount of memory of a given type that is available, as well as to initiate, consult and modify its contents.

Apart from modelling the above mentioned components, the simulator also allows an assembly source file to be indicated. Instead of implementing its own assembler, a more versatile and powerful option has been chosen: ARMSim executes an ARM assembler (e.g., the GNU GCC compiler). Then, it decodes the object code file generated by the ARM assembler to initialize the ROM and the RAM models. At the same time, it also extracts the information about which lines of the source code have generated each machine instruction. In particular, ARMSim follows the specifications and execution algorithms described in the ARMv7-M Architecture Reference Manual[15].

As stated previously, ARMSim does not provide a graphic interface. Instead, when ARMSim is executed, it listens on the port indicated in the command line. In order to interact with ARMSim, it is necessary to establish a connection with that port (e.g., using the `telnet` application) and to use text commands to indicate which actions are to be taken.

The syntax of the commands recognized by the ARMSim simulation engine is described in Figure 1. As it can be seen on that figure, the commands accepted by ARMSim allows the user to: i) assemble a file; ii) consult and modify the content of registers and memory addresses; iii) disassemble memory addresses; iv) define, remove, and consult breakpoints; v) execute code from a given position until the program is finished or until a breakpoint is reached; and vi) execute the program step by step (entering or not into subroutines).

4 | QTARMSIM

As has already been mentioned, QtARMSim is the graphic interface for the ARMSim simulation engine. The relationship between the graphic interface, QtARMSim, which is described in this section, the simulation engine, ARMSim, described in the previous section, and the GNU GCC compiler are schematically presented in Figure 2. As can be seen in that figure, the graphic interface, QtARMSim, takes care of editing the assembly source file and delegates the simulation of the object code to the simulation engine, ARMSim. The communication between the graphic interface and the simulation engine is done using the telnet protocol. The graphic interface sends the actions which the simulation engine has to undertake and as a reply, receives the results of these same actions. The other interaction which takes place, occurs between the simulation engine and the GNU GCC compiler. The simulation engine, when it is required for this, executes the `gcc` command in a suitable way, in order to assemble the source code indicated by the graphic interface. Once the the source code has been assembled, the simulation engine loads the resulting object code into the memory, informs the graphic interface of its new state and then waits for further requests.

In addition to the explained above, the graphic interface is also responsible for starting the simulation engine and to manage its configuration —which port it should listen to, the `gcc` command path, and its compilation parameters—.

It goes without saying that all the previously described interactions are conveniently hidden from the graphic interface user. When QtARMSim is executed, the user simply interacts with a graphic window like the one shown in Figure 3. QtARMSim presents two working modes: the edit mode and the simulation mode. Figure 3 shows the appearance of the QtARMSim main

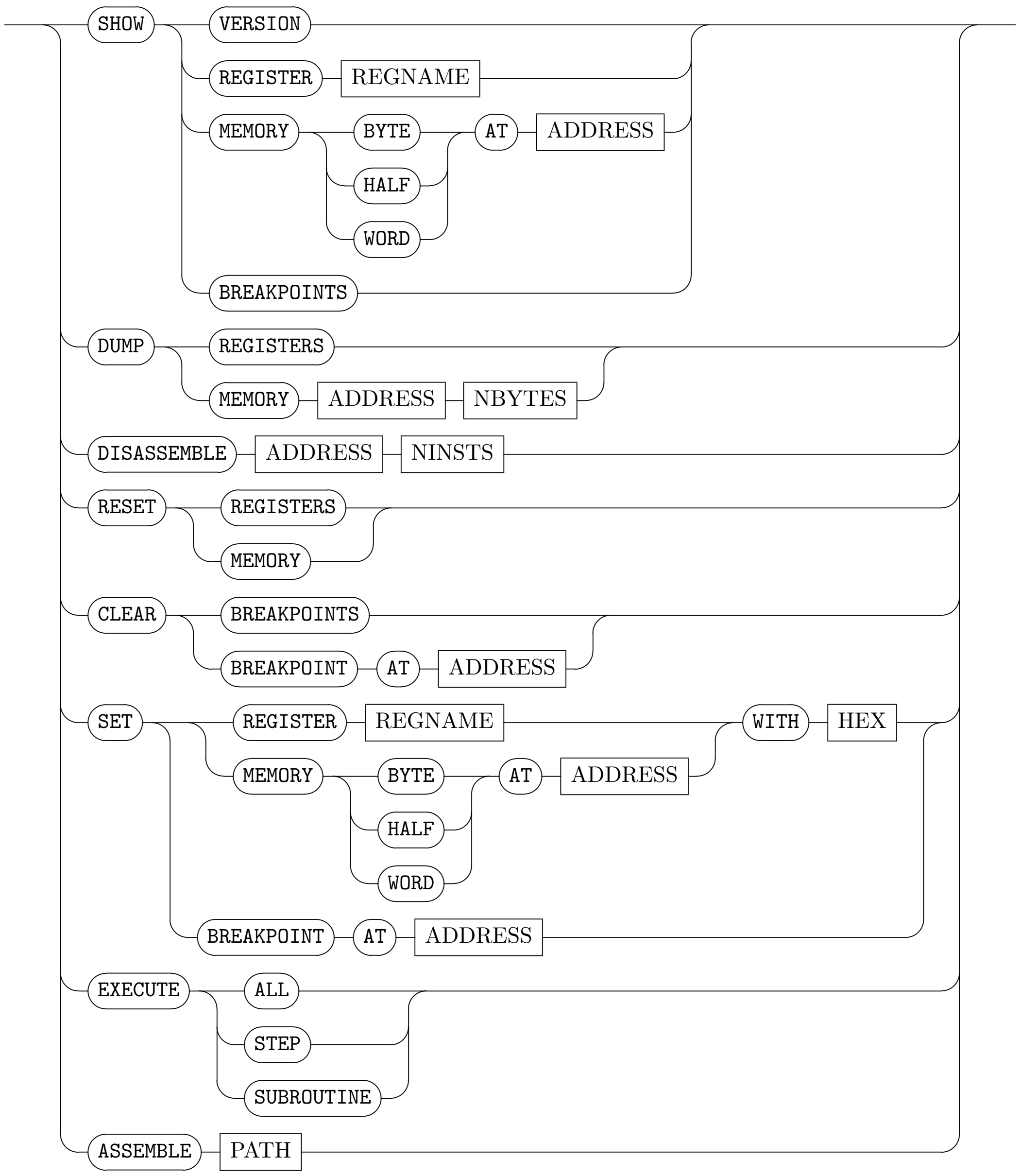


FIGURE 1 Syntax of the commands recognized by the ARMSim simulation engine

window when editing a small code can be seen (thus, when in edit mode). In this mode, the central part of the window corresponds to the editor of the assembly source code. Around this central part, a series of panels is distributed. In the default layout, the registers panel is located to the left of the editor, and the memory panel to its right. Below, another section is displayed with three tabbed panels: a message panel, a memory dump panel, and an LCD display panel. The latter two being new additions compared to the simulator presented at[13].

As can be seen in Figure 3, the memory and registers panels are disabled when in the edit mode —they will be described on Section 4.3—. As far as the message panel is concerned, it has the function of showing those messages related to the actions which are being done. For example, whether the source code has been correctly assembled or not, which line has just been executed, etc. Finally, the memory dump and LCD Displays panels, which are used when in simulation mode, will be described on Section 4.3.

As all the panels are implemented as dockable widgets, it is possible to show, hide, resize, detach or stack them. This feature allows the user to fully customize how the information is being presented at any given moment. As an example, a different

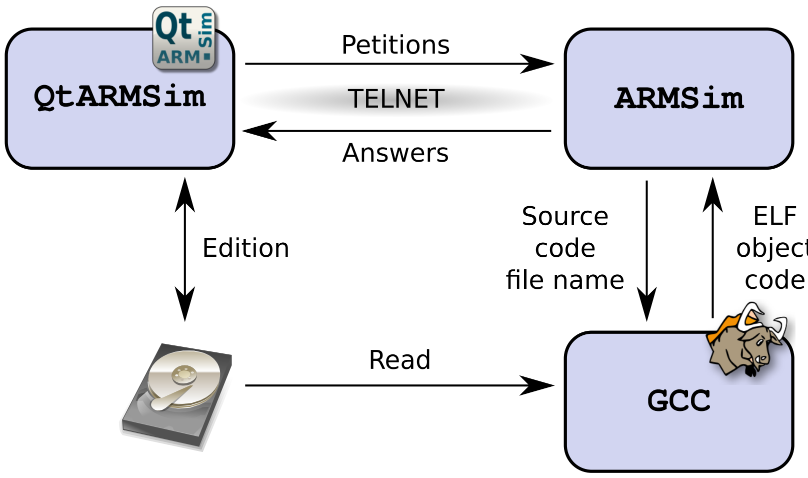


FIGURE 2 Components of the simulator and their interactions

layout, where the registers and memory panels have been closed, can be seen on Figure 6. In addition to this, a convenient *Restore Default Layout* menu entry is provided to restore the default layout. This option is specially handy when a student has involuntarily messed up the simulator layout in a laboratory session, as it allows the teacher to easily restore the default layout.

Compared to the version described in[13], the current version of QtARMSim is bundled with the GNU GCC ARM binary files required to assemble an ARM source code on GNU/Linux, Microsoft Windows, and Mac OS X. This bundling simplifies both the QtARMSim installation process and the automatic GNU GCC path configuration (see Section 4.1). Also, as only those files of the GNU GCC toolchain that are strictly required to assemble are included, it also uses much less space than would be required if the whole GNU GCC for ARM toolchain were to be manually installed by the user.

On the next subsections, the QtARMSim configuration, and its edit and simulation modes will be described.

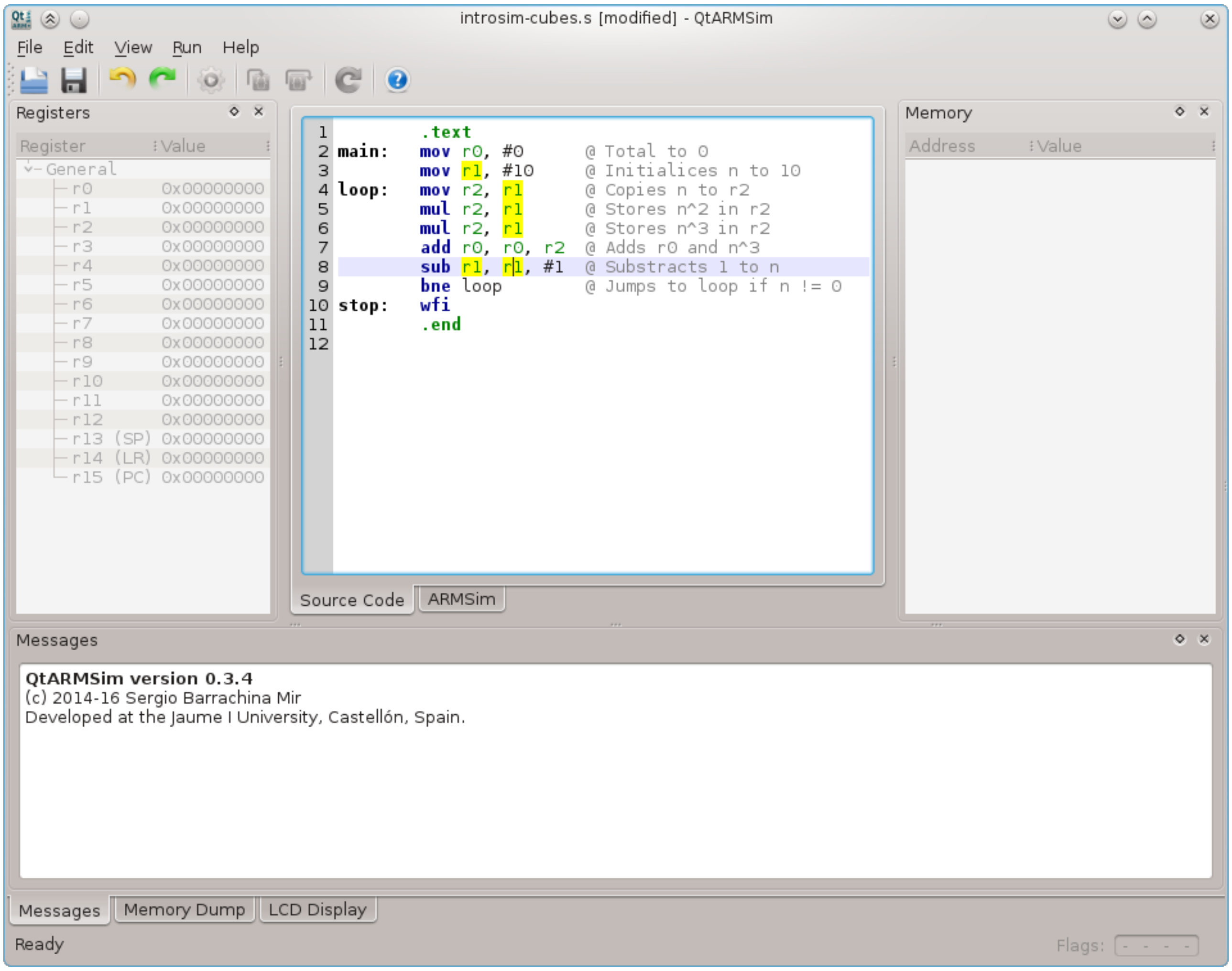
4.1 | Configuration

On its first execution, QtARMSim will try to automatically find out all its necessary parameters. Therefore, in most cases, the user will not have to manually modify the QtARMSim configuration. Nevertheless, it is possible to manually configure how to execute the ARMSim simulation engine or where the GNU GCC cross compiler for ARM is installed. These options are entered on the preferences dialog box of QtARMSim (see Figure 4). The top frame of the configuration dialog box corresponds to those parameters related to the ARMSim simulation engine and allows to configure: the server and port to which the graphic interface must be connected; the command line for executing the simulation engine (in the case that it is executed on the same machine as the interface); and its working directory. The bottom frame manages the configuration related to the GNU GCC compiler for ARM. This frame can be used to indicate the route to the GNU GCC compiler for ARM executable and the command line options that should be used to assemble the source code.

It should be noted than the configuration dialog box provides a *Restore Defaults* button that allows to restore the automatically obtained configuration. As mentioned above about the *Restore Default Layout* menu option, this button is specially handy when in a laboratory class context, as it provides the teacher a fast mechanism to recover the QtARMSim default configuration in case an student has messed it up by mistake.

4.2 | Edit mode

In edit mode, as has been mentioned, the central part of the window is an ARM assembly source code editor, which allows the student to write the assembly program which he wishes to subsequently simulate. Figure 3, referenced above, shows the QtARMSim window in which a small assembly program is being edited. As can be seen in the illustration, the editor recognizes the ARM Thumb assembly language and conveniently highlights the source code. In addition to this syntax highlighting, the source code editor also presents the following features: i) it allows the font size to be changed, which is very helpful when doing classroom presentations; and ii) both registers and labels in the source code are recognized, so that other occurrences of the

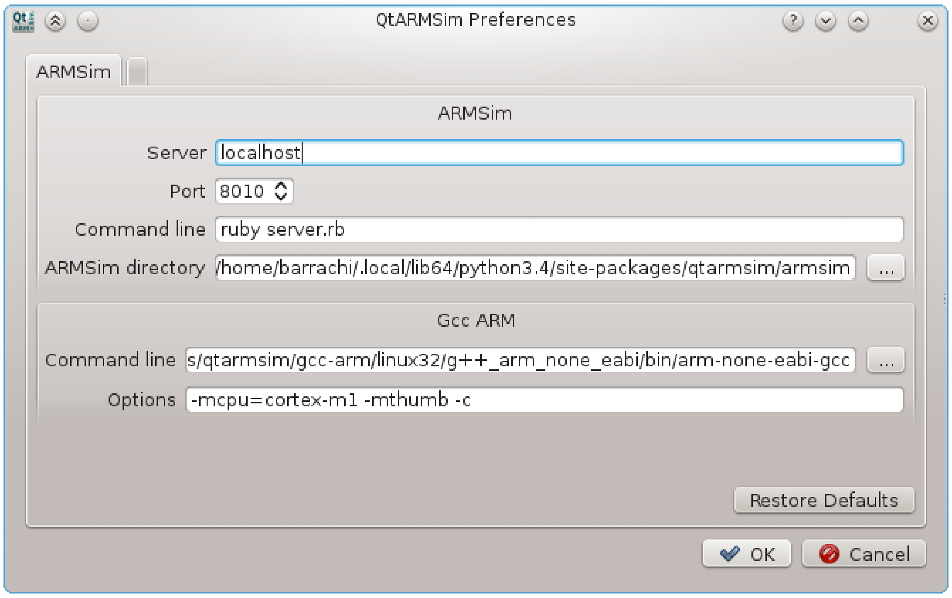


Register	Value
General	
r0	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13 (SP)	0x00000000
r14 (LR)	0x00000000
r15 (PC)	0x00000000

```
        .text
main:   mov r0, #0      @ Total to 0
        mov r1, #10     @ Initialices n to 10
loop:   mov r2, r1      @ Copies n to r2
        mul r2, r1      @ Stores n^2 in r2
        mul r2, r1      @ Stores n^3 in r2
        add r0, r0, r2  @ Adds r0 and n^3
        sub r1, r1, #1  @ Substracts 1 to n
        bne loop        @ Jumps to loop if n != 0
stop:   wfi
        .end
```

QtARMSim version 0.3.4
(c) 2014-16 Sergio Barrachina Mir
Developed at the Jaume I University, Castellón, Spain.

FIGURE 3 QtARMSim main window in edit mode



ARMSim

Server	localhost
Port	8010
Command line	ruby server.rb
ARMSim directory	/home/barrachi/.local/lib64/python3.4/site-packages/qtarmsim/armsim

Gcc ARM

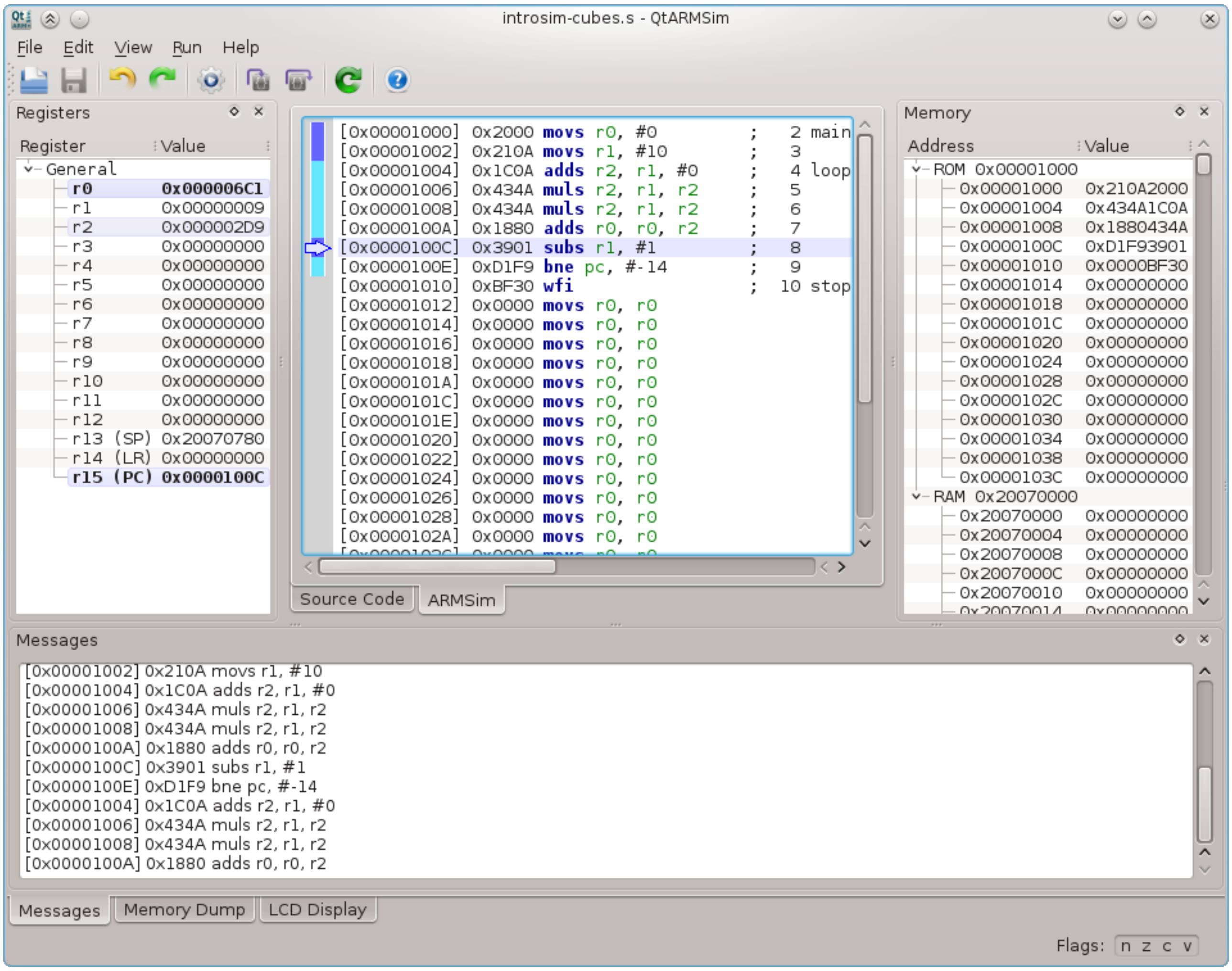
Command line	s/qtarmsim/gcc-arm/linux32/g++_arm_none_eabi/bin/arm-none-eabi-gcc
Options	-mcpu=cortex-m1 -mthumb -c

FIGURE 4 QtARMSim preferences dialog box

currently focused register or label are highlighted, which makes it easier to follow data dependencies during the development or debugging of the code. One example of this last characteristic can be seen in Figure 3: as the text cursor is placed above an appearance of the `r1` register, every occurrence of this register in the source code is highlighted in yellow. The same will happen when the text cursor is on a label.

4.3 | Simulation Mode

In order to go into simulation mode it is only necessary to click on the tab labeled with "ARMSim" —this tab can be found below the central section of the main window—. The appearance of QtARMSim when it is in simulation mode can be seen in Figure 5.



```
Register        Value
General
  r0            0x000006C1
  r1            0x00000009
  r2            0x000002D9
  r3            0x00000000
  r4            0x00000000
  r5            0x00000000
  r6            0x00000000
  r7            0x00000000
  r8            0x00000000
  r9            0x00000000
  r10           0x00000000
  r11           0x00000000
  r12           0x00000000
  r13 (SP)      0x20070780
  r14 (LR)      0x00000000
  r15 (PC)      0x0000100C
```

```
[0x00001000] 0x2000 movs r0, #0      ;  2 main
[0x00001002] 0x210A movs r1, #10     ;  3
[0x00001004] 0x1C0A adds r2, r1, #0  ;  4 loop
[0x00001006] 0x434A muls r2, r1, r2  ;  5
[0x00001008] 0x434A muls r2, r1, r2  ;  6
[0x0000100A] 0x1880 adds r0, r0, r2  ;  7
[0x0000100C] 0x3901 subs r1, #1      ;  8
[0x0000100E] 0xD1F9 bne pc, #-14     ;  9
[0x00001010] 0xBF30 wfi              ; 10 stop
[0x00001012] 0x0000 movs r0, r0
[0x00001014] 0x0000 movs r0, r0
[0x00001016] 0x0000 movs r0, r0
[0x00001018] 0x0000 movs r0, r0
[0x0000101A] 0x0000 movs r0, r0
[0x0000101C] 0x0000 movs r0, r0
[0x0000101E] 0x0000 movs r0, r0
[0x00001020] 0x0000 movs r0, r0
[0x00001022] 0x0000 movs r0, r0
[0x00001024] 0x0000 movs r0, r0
[0x00001026] 0x0000 movs r0, r0
[0x00001028] 0x0000 movs r0, r0
[0x0000102A] 0x0000 movs r0, r0
```

```
Address          Value
ROM 0x00001000
  0x00001000     0x210A2000
  0x00001004     0x434A1C0A
  0x00001008     0x1880434A
  0x0000100C     0xD1F93901
  0x00001010     0x0000BF30
  0x00001014     0x00000000
  0x00001018     0x00000000
  0x0000101C     0x00000000
  0x00001020     0x00000000
  0x00001024     0x00000000
  0x00001028     0x00000000
  0x0000102C     0x00000000
  0x00001030     0x00000000
  0x00001034     0x00000000
  0x00001038     0x00000000
  0x0000103C     0x00000000
RAM 0x20070000
  0x20070000     0x00000000
  0x20070004     0x00000000
  0x20070008     0x00000000
  0x2007000C     0x00000000
  0x20070010     0x00000000
```

```
[0x00001002] 0x210A movs r1, #10
[0x00001004] 0x1C0A adds r2, r1, #0
[0x00001006] 0x434A muls r2, r1, r2
[0x00001008] 0x434A muls r2, r1, r2
[0x0000100A] 0x1880 adds r0, r0, r2
[0x0000100C] 0x3901 subs r1, #1
[0x0000100E] 0xD1F9 bne pc, #-14
[0x00001004] 0x1C0A adds r2, r1, #0
[0x00001006] 0x434A muls r2, r1, r2
[0x00001008] 0x434A muls r2, r1, r2
[0x0000100A] 0x1880 adds r0, r0, r2
```

FIGURE 5 QtARMSim in simulation mode after executing some instructions

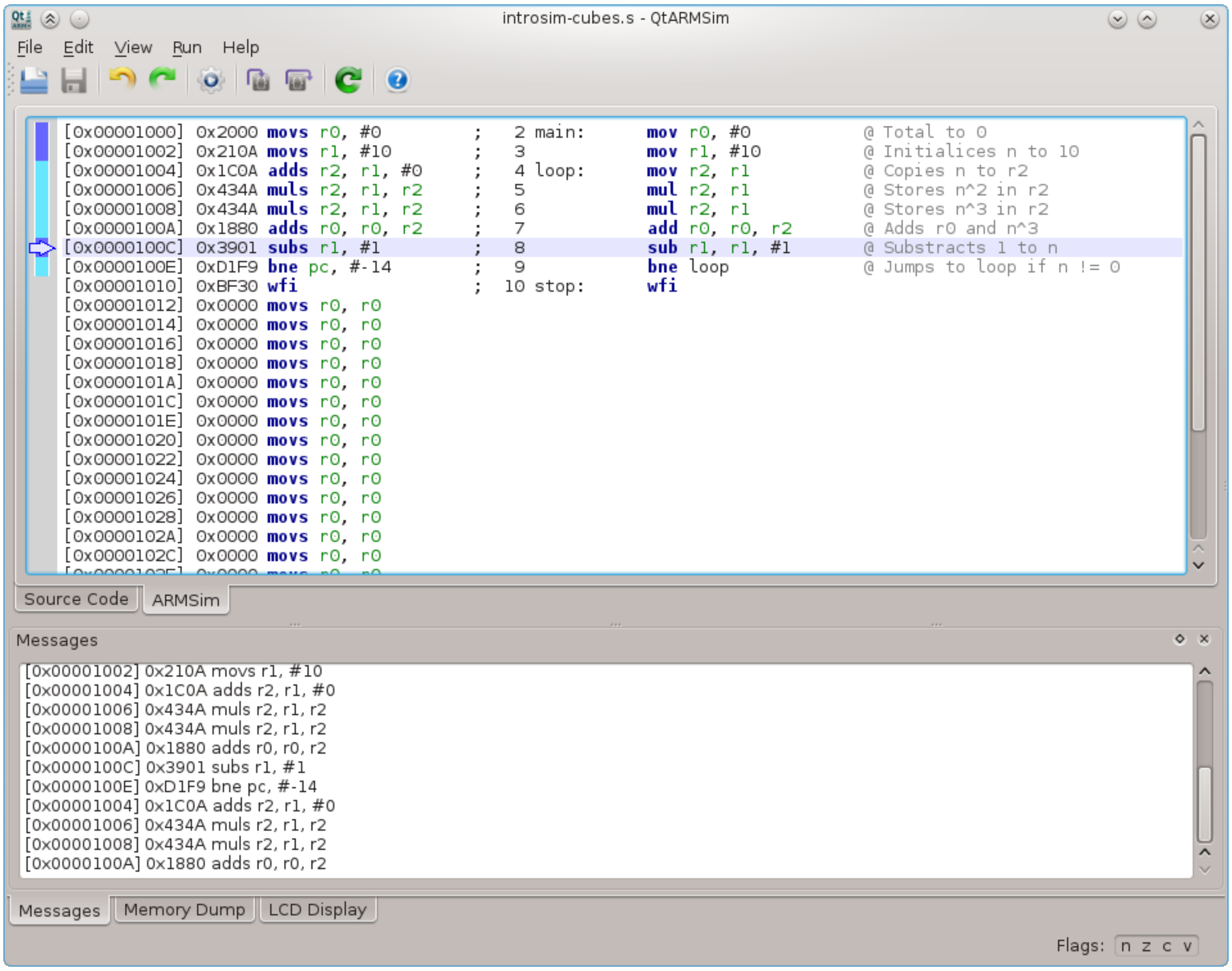
When changing from edit to simulation mode, behind the scenes, the graphic interface instructs the ARMSim simulation engine to perform the following actions: i) execute the cross assembler to assemble the source code and create an object code file; ii) create as memory regions as the ones indicated by the object code file; iii) initialize the ROM memories with the machine instructions indicated by the object code file; iv) initialize, if this is the case, the RAM memories with the data indicated by the object code file; and lastly, v) initialize the registers of the simulated processor. If no error occurs, the change to the simulation mode is completed. Otherwise, a dialogue box is shown which informs that an error has occurred, detailed error information is printed on the messages panel, and the change to the simulation mode is aborted.

Once in the simulation mode, every line of the central window shows the information corresponding to a machine instruction (see Figure 5). For each machine instruction, it is shown (from left to right):

1. The memory address where the machine instruction is stored.
2. The machine instruction expressed in hexadecimal.
3. The machine instruction decoded in assembly language.
4. The original line in the assembly source code which has produced this machine instruction.

So for example, the information shown in the first line of the simulator window on Figure 6 indicates that:

1. The machine instruction is stored in the memory address `0x0000 1000`.
2. The machine instruction expressed in hexadecimal is `0x2000`.
3. The machine instruction decoded in assembly language is “`movs r0, #0`”.
4. The instruction has been generated from the second line of the assembly source code, which is: “`main: mov r0, #0 @ Total to 0`”.



```
[0x00001000] 0x2000 movs r0, #0      ;   2 main:    mov r0, #0       @ Total to 0
[0x00001002] 0x210A movs r1, #10     ;   3          mov r1, #10      @ Initialices n to 10
[0x00001004] 0x1C0A adds r2, r1, #0  ;   4 loop:    mov r2, r1       @ Copies n to r2
[0x00001006] 0x434A muls r2, r1, r2  ;   5          mul r2, r1       @ Stores n^2 in r2
[0x00001008] 0x434A muls r2, r1, r2  ;   6          mul r2, r1       @ Stores n^3 in r2
[0x0000100A] 0x1880 adds r0, r0, r2  ;   7          add r0, r0, r2   @ Adds r0 and n^3
[0x0000100C] 0x3901 subs r1, #1      ;   8          sub r1, r1, #1   @ Substracts 1 to n
[0x0000100E] 0xD1F9 bne pc, #-14     ;   9          bne loop         @ Jumps to loop if n != 0
[0x00001010] 0xBF30 wfi              ;  10 stop:    wfi
[0x00001012] 0x0000 movs r0, r0
[0x00001014] 0x0000 movs r0, r0
[0x00001016] 0x0000 movs r0, r0
[0x00001018] 0x0000 movs r0, r0
[0x0000101A] 0x0000 movs r0, r0
[0x0000101C] 0x0000 movs r0, r0
[0x0000101E] 0x0000 movs r0, r0
[0x00001020] 0x0000 movs r0, r0
[0x00001022] 0x0000 movs r0, r0
[0x00001024] 0x0000 movs r0, r0
[0x00001026] 0x0000 movs r0, r0
[0x00001028] 0x0000 movs r0, r0
[0x0000102A] 0x0000 movs r0, r0
[0x0000102C] 0x0000 movs r0, r0
```

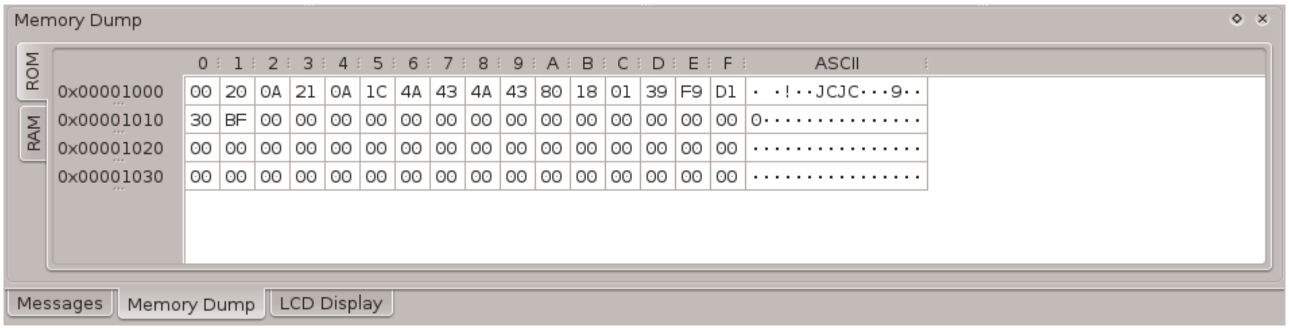
```
[0x00001002] 0x210A movs r1, #10
[0x00001004] 0x1C0A adds r2, r1, #0
[0x00001006] 0x434A muls r2, r1, r2
[0x00001008] 0x434A muls r2, r1, r2
[0x0000100A] 0x1880 adds r0, r0, r2
[0x0000100C] 0x3901 subs r1, #1
[0x0000100E] 0xD1F9 bne pc, #-14
[0x00001004] 0x1C0A adds r2, r1, #0
[0x00001006] 0x434A muls r2, r1, r2
[0x00001008] 0x434A muls r2, r1, r2
[0x0000100A] 0x1880 adds r0, r0, r2
```

FIGURE 6 QtARMSim in simulation mode, with the registers and memory panels closed

The contents of the `r1` to `r15` registers are shown on the registers panel (see left part of Figure 5). The `r15` register requires special mention since in the ARM architecture, it is the *Program Counter* (`PC`). QtARMSim highlights the line that corresponds to the address indicated by the `PC`, in order to help the student to identify which line of code is the next one to be executed.

Furthermore, the memory content of the simulated computer is shown on the memory panel, as can be seen on the right part of Figure 5. In this example, it can be noticed that the simulated computer has two memory blocks: a ROM memory block, which begins at address `0x0000 1000`, and a RAM memory block, which begins at address `0x2007 0000`. It can also be seen that some ROM memory cells have nonzero values, which correspond to the machine code generated from the assembled source code, and, that in this particular example, all the RAM cells are set to zero. QtARMSim allows the student to manually edit the contents of any RAM cell by double clicking over it and entering a numeric value (in decimal, hexadecimal, or binary), or a string. If the value entered exceeds the word capacity, an error is generated and an explanation is given to the student.

The current version of QtARMSim has incorporated two new panels: a memory dump panel and an LCD display panel. The memory dump panel shows a more compact representation of the memory (see Figure 7). In this panel, each memory block is displayed in a different tab, and in each tab, the contents of the corresponding memory block are represented at the byte level, 16 bytes per row. In addition to the hexadecimal representation of each byte, an ASCII representation of all the bytes in a row is also shown in the last column. This panel is particularly useful to understand the Little Endian memory organization, since it makes it possible to compare the memory contents represented at a byte level with the same contents grouped by words in the memory panel. It also helps the students to understand that the same binary value can be used to represent different types of data. And finally, to properly work with examples where vectors, strings or characters are involved. As was the case in the memory panel, the student can manually edit the contents of any RAM byte entering a numeric vale (in decimal, hexadecimal, or binary), or a character. Again, if the value entered exceeds the byte capacity, an error is generated and an explanation is given to the student.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
0x00001000	00	20	0A	21	0A	1C	4A	43	4A	43	80	18	01	39	F9	D1	· ·!··JCJC···9··
0x00001010	30	BF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0···············
0x00001020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	················
0x00001030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	················

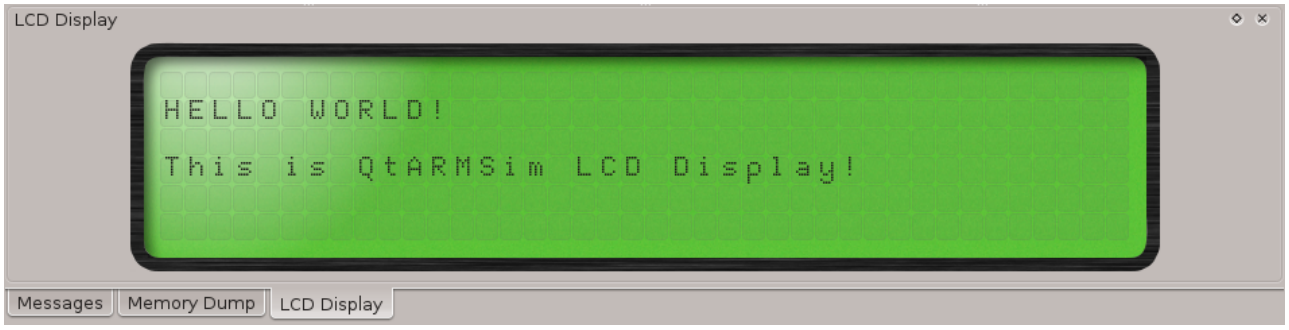
FIGURE 7 QtARMSim memory dump panel

The second panel that has been added to QtARMSim is an LCD display (see Figure 8). This panel mimics a 40×6 LCD display which is mapped to a RAM memory region. It displays each byte in that region as if it was an ASCII character. The purpose of this LCD display is to provide the students a first approximation to the input/output problem and to allow their programs to display results in an visible way. Moreover, as each LCD character is actually a byte of a matrix that is stored in memory, it also provides them with a playground to check how the address of a matrix element should be computed.

Once the different graphic elements available on the simulation mode have been described, the different actions that the student can perform will be described in the next subsections: i) execute the whole program, ii) execute the program step by step, iii) edit the contents of registers and memory, iv) define execution breakpoints.

4.3.1 | Full program execution

The simplest simulation action is to execute the whole program. Figure 9 displays the QtARMSim window after executing the machine code corresponding to the source file “add.s”, which adds the bytes at addresses `0x2007 0000` and `0x2007 0001`, and writes the result on the byte at `0x2007 0002`. As you can see in the said figure, QtARMSim highlights those registers and memory positions which have been written at any time of the code execution. Thus, as it can be seen in Figure 9, after executing the shown machine code, the following elements, those that have been written by the program, are in bold and have a blue background:



```
HELLO WORLD!
This is QtARMSim LCD Display!
```

FIGURE 8 QtARMSim LCD display panel

i) the `r0`, `r1`, and `r15` (`PC`) registers; ii) the word at `0x2007 0000` in the memory panel; and iii) the byte at `0x2007 0002` in the memory dump panel. This highlighting feature has been implemented in order to allow the students to easily identify those registers and memory positions that have been written during the code execution.

As the current version of QtARMSim, compared to the version described in[13], provides a new visualization of the memory at the byte level, which is shown on the new memory dump panel, the graphic interface memory model had to be completely rewritten to work at this lower level. Taking advantage of this major modification, the history of which memory positions have been written has also been narrowed down to the byte level. This way, although the memory panel highlights those words where any of its bytes have been rewritten, the memory dump panel can be used to see which bytes of those words have been actually rewritten. The aforementioned Figure 9 illustrated this feature. As stated before, when the machine code is executed, the byte at address `0x2007 0002` is written with the computed result. Therefore, the whole word at `0x2007 0000` is highlighted on the memory panel, whereas only the byte at address `0x2007 0002` is highlighted on the memory dump panel.

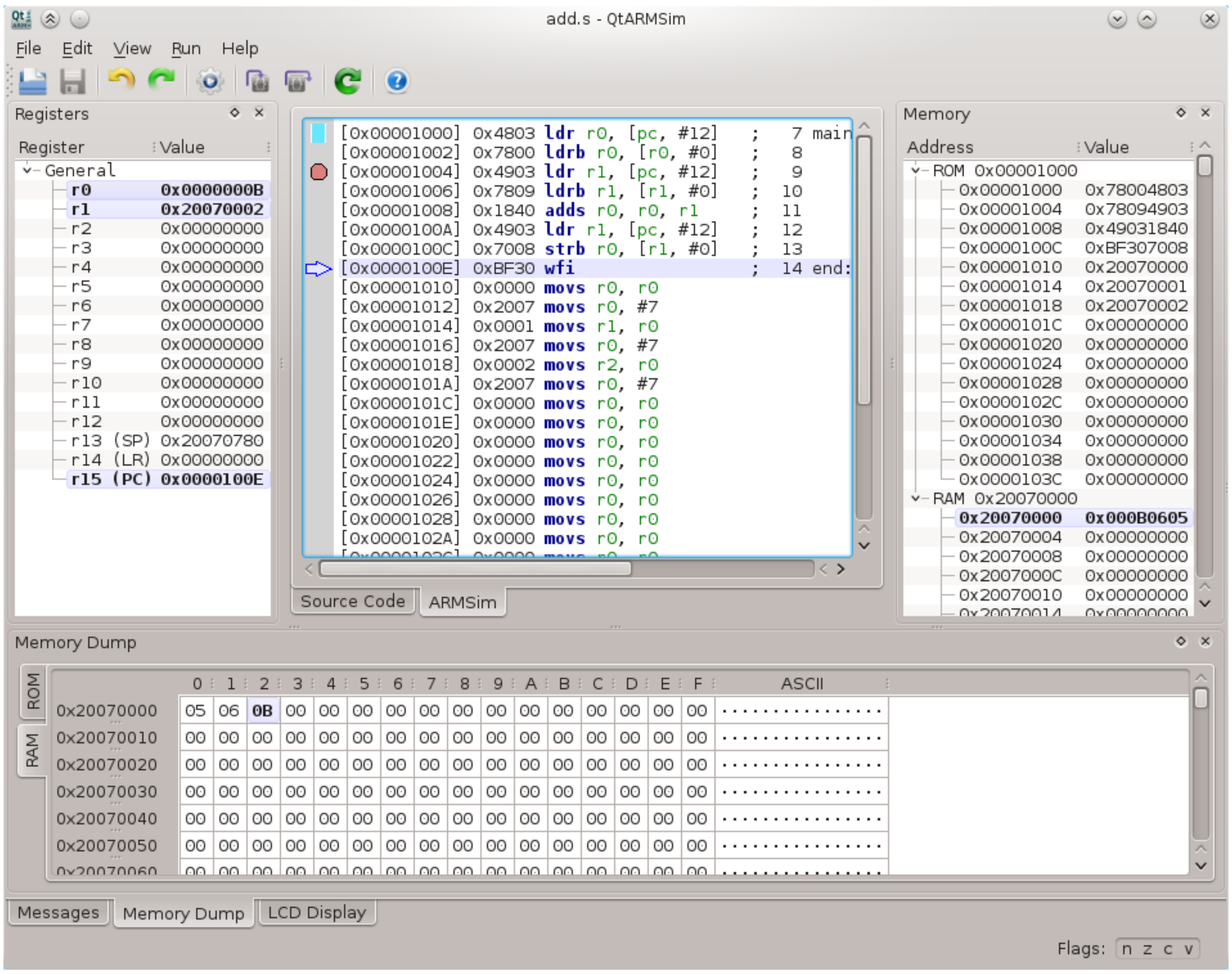
4.3.2 | Step by step execution

Although the whole execution of a program, described on the previous subsection, can serve to test if the program does what is supposed to do, it does not allow the student to see in detail how the final state has been reached. It only allows to compare the initial state of the simulated computer with its final state after executing the whole program.

In order to evaluate what happens when executing each machine instruction, the simulator provides an option to execute the machine code step by step (i.e., machine instruction by machine instruction). This option allows the student to become familiar with what each machine instruction actually does. It is also used in practice to see why a program or one of its parts is not doing what is expected of it. Moreover, this option permits the student to see how a modification of either a register or a memory position at any given moment of the execution alters the subsequent execution.

The simulator provides two step by step execution modes. The first one, called *step into*, executes only one machine instruction at each step. The second one, called *step over*, takes into account that a program is usually structured using subroutines, and executes one machine instruction at each step except when a call to a subroutine is found. In this latter case, it executes the whole subroutine as if it was a unique machine instruction. This second option helps the student to compare the state of the simulated computer before calling a subroutine and after executing it.

A new feature of the current version of QtARMSim is that each time a step by step execution is done, a colored ribbon is drawn at the left margin of the last machine instruction executed. While the instructions are executed in sequential order, the color of the ribbon is maintained, but when a jump is taken, a new color for the ribbon is chosen —that will remain the same until another jump is made—. This way, the student can visually follow which machine instructions have been already executed, and also can visually see the effect of control flow instructions (i.e., conditional and unconditional branches, for and while structures, etc.). This color ribbon can be seen on Figures 5 and 6. On these two figures, the machine code has been executed until the moment in which its loop is going to be finished for the second time. Figure 10 illustrates how the margin decoration changed when the machine code shown on Figures 5 and 6 were executed step by step. The left snapshot of Figure 10 shows the left margin at the start of the simulation, when no instruction was executed yet, and the PC was pointing at the first instruction. The right snapshot corresponds to the margin as depicted on Figures 5 and 6, were twelve instructions have been executed. As it can be seen, when the eighth instruction, `bne pc, #-14`, was executed, and due to the sequential order execution being broken, the



```
Register     Value
General
  r0         0x0000000B
  r1         0x20070002
  r2         0x00000000
  r3         0x00000000
  r4         0x00000000
  r5         0x00000000
  r6         0x00000000
  r7         0x00000000
  r8         0x00000000
  r9         0x00000000
  r10        0x00000000
  r11        0x00000000
  r12        0x00000000
  r13 (SP)   0x20070780
  r14 (LR)   0x00000000
  r15 (PC)   0x0000100E
```

```
[0x00001000] 0x4803 ldr r0, [pc, #12]   ;  7 main
[0x00001002] 0x7800 ldrb r0, [r0, #0]   ;  8
[0x00001004] 0x4903 ldr r1, [pc, #12]   ;  9
[0x00001006] 0x7809 ldrb r1, [r1, #0]   ; 10
[0x00001008] 0x1840 adds r0, r0, r1     ; 11
[0x0000100A] 0x4903 ldr r1, [pc, #12]   ; 12
[0x0000100C] 0x7008 strb r0, [r1, #0]   ; 13
[0x0000100E] 0xBF30 wfi                 ; 14 end:
[0x00001010] 0x0000 movs r0, r0
[0x00001012] 0x2007 movs r0, #7
[0x00001014] 0x0001 movs r1, r0
[0x00001016] 0x2007 movs r0, #7
[0x00001018] 0x0002 movs r2, r0
[0x0000101A] 0x2007 movs r0, #7
[0x0000101C] 0x0000 movs r0, r0
[0x0000101E] 0x0000 movs r0, r0
[0x00001020] 0x0000 movs r0, r0
[0x00001022] 0x0000 movs r0, r0
[0x00001024] 0x0000 movs r0, r0
[0x00001026] 0x0000 movs r0, r0
[0x00001028] 0x0000 movs r0, r0
[0x0000102A] 0x0000 movs r0, r0
```

```
Address          Value
ROM 0x00001000
  0x00001000     0x78004803
  0x00001004     0x78094903
  0x00001008     0x49031840
  0x0000100C     0xBF307008
  0x00001010     0x20070000
  0x00001014     0x20070001
  0x00001018     0x20070002
  0x0000101C     0x00000000
  0x00001020     0x00000000
  0x00001024     0x00000000
  0x00001028     0x00000000
  0x0000102C     0x00000000
  0x00001030     0x00000000
  0x00001034     0x00000000
  0x00001038     0x00000000
  0x0000103C     0x00000000
RAM 0x20070000
  0x20070000     0x000B0605
  0x20070004     0x00000000
  0x20070008     0x00000000
  0x2007000C     0x00000000
  0x20070010     0x00000000
```

```
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   ASCII
0x20070000   05 06 0B 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x20070010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x20070020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x20070030   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x20070040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x20070050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

FIGURE 9 QtARMSim in simulation mode after executing the whole program

ribbon color changed from blue to light blue. It can also be seen that the new light blue color was kept while the next instructions were executed in sequential order.

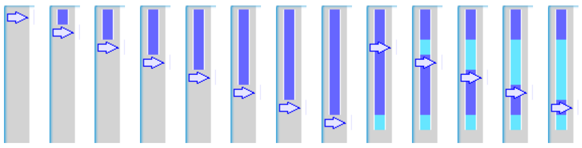


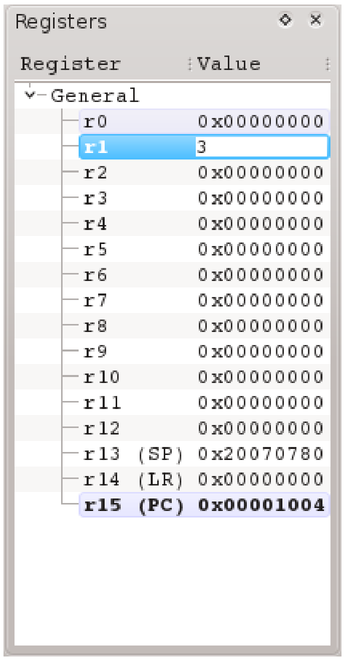
FIGURE 10 Detail of how the left margin decoration changes as the machine code shown on Figures 5 and 6 is executed step by step —whenever a jump is done, the ribbon color changes—

4.3.3 | Edit the contents of registers and memory

As mentioned in the previous subsection, one of the uses of the step by step execution is to manually modify the contents of a register or a memory position and see how this modification alters the subsequent execution.

QtARMSim makes it possible to modify the contents of a register or a memory position by double clicking on the cell whose content is to be modified, and writing the new contents followed by the Enter key. As an example, Figure 11 shows how the contents of the `r1` register are going to replaced by the number 3. The new contents can be either a numeric value or a string. If it is a numeric value, it can be entered either on decimal, as is, on hexadecimal, using the "0x" prefix, on octal, using the "0" prefix, or on binary, using the "0b" prefix. On the other hand, if the new contents is a string, or a character, it should be entered either quoted or double quoted (e.g., 'Hi' or "Hi").

Another new feature of the current version of QtARMSim is that when the entered value can not be represented on the chosen register or memory position, due to its binary size being greater than the one of the chosen holder, either a word or a byte, the user gets a message pointing out this error. This feature can be used to stress that a given value can only be stored if there are enough bits to represent it.



Register	Value
General	
r0	0x00000000
r1	3
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13 (SP)	0x20070780
r14 (LR)	0x00000000
r15 (PC)	**0x00001004**

FIGURE 11 Manually editing the contents of the `r1` register

4.3.4 | Breakpoints

The previously described step by step execution allows us to look carefully at what is actually happening in one specific point of the code. However, in order to get to the area that we want to inspect in detail, we may have to execute several instructions first. For example, we might be interested in a part of the code that can only been reached after a loop with hundreds of iterations. It would not make sense to go step by step until we manage to get out of the loop and arrive to the point of the code that we really want to see in detail. This is one case where using a breakpoint to instruct the simulator where it should stop executing instructions comes in handy. Setting a breakpoint will allow the simulator to execute those parts of the code that we are not interested in seeing in detail and stop the execution in the machine instruction from which the step by step will be done.

Other situation in which setting breakpoints can also be useful is to easily observe the simulator state evolution at different points of a program.

QtARMSim makes it possible to set and unset breakpoints with ease. To set a breakpoint, it is enough to click on the left hand margin of the line in which we want to set the breakpoint (on the simulation window). In doing this, a red symbol will appear in the margin to indicate that a breakpoint has been set in that line. To illustrate this, Figure 9 shows a breakpoint set at the memory address `0x0000 1004`. On the other hand, to unset a previously created breakpoint, it is only necessary to click on the breakpoint mark that has to be unset.

5 | RESULTS

An ARM simulator that complies with the prescribed objectives in Section 2 has been developed and can be downloaded free of charge. The version of the simulator presented in this paper, with respect to the version published in Spanish in[13], implements, among other improvements and enhancements: i) a color trace of the instructions that have been simulated, ii) a memory dump at a byte level, iii) a basic LCD display device simulation, and iv) a bundled GCC assembler. These new enhancements provide the student a better understanding of the instruction execution flow and the memory organization, and an easier installation and configuration of the simulator.

It has also been written up and published under a *Creative Commons* license a textbook, in Spanish, that has been used and updated during the courses 2014 and 2015. In this textbook, the simulator is described with more detail, and laboratory exercises are provided that make an extensive use of the proposed simulator. In particular, the textbook relies on ARMSim and QtARMSim to cover the following topics of the *Fundamentals of computer architecture* knowledge unit defined at the Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering[8]:

- Organization of the von Neumann machine.
- Instruction formats.
- The fetch/execute cycle; instruction decoding and execution.
- Registers.
- Instruction types and addressing modes.
- Subroutine call and return mechanisms.
- Programming in assembly language.

Since the courses of 2014 and 2015 have been the first in which the simulator has been used, students have been constantly asked for their opinions on its usability, so we received invaluable feedback on those aspects that should be improved, which allowed us to constantly improve the simulator and reach its current state.

In general, the feedback has been very positive, the sharpest criticism being the installation process, mostly due to the fact that the simulator relies on a series of dependencies that can be awkward to install on one operating system in particular. To address this problem, we have already lightened the installation process in two ways. A first step consisted on migrating the graphic interface dependencies from PyQt and QScintilla to PySide and our own code editor implementation. This alleviated the installation procedure because the PySide dependency is easier to install than the PyQt and QScintilla ones. A second step has been to bundle the indispensable GNU GCC binary files in our package, so that this dependency does not have to be manually installed by the student, and the GNU GCC path configuration option can be automatically set.

It should be highlighted that the feedback from repeaters who on the previous course did their training with MIPS and the next course had to change to ARM, has been especially gratifying. Their perception on the usability of the simulator was so good that they commented us that the assembly language of ARM was much simpler than MIPS (when in fact we do not share this opinion).

In addition to the previous points, on the first year the simulator was used, we conducted a survey with the following items:

1. I have installed QtARMSim in my computer without any problem.
2. The graphic interface of QtARMSim seems adequate.
3. The code edition has been easy for me.
4. The debug options (step by step execution, breakpoints, etc.) have been useful for me.
5. The information displayed on the different panels of QtARMSim is adequate.
6. I consider that the simulator has helped me understand better how the ARM processor works.
7. I would recommend that we continue using QtARMSim in Computer Architecture teaching.

8. I consider that this year's course using QtARMSim has been much more interesting than last years with the xspim simulator.
9. What positive features would you highlight about QtARMSim?
10. What do you think needs to be done to improve QtARMSim?

The first eight questions were Likert items ranged from 1 to 5, being 1 "disagree strongly", and 5, "agree strongly". The last two question were open questions. This survey was answered by 28 students (19% of the enrolled ones).

The first five Likert items of the survey were about the technical aspects of the simulator (installation, interface, debugging and provided information). In those questions, a mean score of 4 was obtained, except for the installation one, where the mean was 3.

The following three Likert items, more related to the teaching objective of the simulator and the student satisfaction, *"I consider that the simulator has helped me understand better how the ARM processor works"*, *"I would recommend that we continue using QtARMSim in Computer Architecture teaching"*, and *"I consider that this year's course using QtARMSim has been much more interesting than last years with the xspim simulator"*, obtained a mean score of 5.

The answers given to the last two questions, the open ones, *"What positive features would you highlight about QtARMSim?"*, and *"What do you think needs to be done to improve QtARMSim?"*, stressed, on one hand, the ease of use of the simulator and how it was helpful to learn the subject. On the other hand, they also pointed out that the installation process needed to be simplified. Some of the responses obtained to these two questions are reproduced in Tables 1 and 2, respectively. It should be noted, as previously stated, that the survey was conducted on the first year that the simulator was in use. In particular, the suggestions for an easier installation and a more automatic configuration, have already been addressed, in the ways explained before.

TABLE 1 Some of the answers to the question "What positive features would you highlight about QtARMSim?" obtained after the first year the simulator was used

Simple and easy to understand.
Easy to use. Detailed information.
It is well designed in all its aspects.
The use of the step by step procedure and the clear position of the flags.
Ease of the interface and its usage. Very useful for studying.
Being able to execute ARM assembly code and to see the contents of the registers and the memory.
It provides a very visual way to see the how the memory and the registers are modified while each instruction is being executed.
It is clear what kind of information, data or a memory address, is stored on each register. It is also very graphic and not confusing, like the previous `xspim` simulator, and what is going on is a lot more understandable.
The simulator graphically represents very well: the execution of ARM programs, where the data is stored, from where it is loaded, what is done with it, etc. Once you understand the theory, the simulator is very visual and helps to eventually understand the course related concepts.

Finally, the laboratory teachers assessment of the simulator has been very positive and their impression has been that the students were more aware of what was happening as they were doing the simulator exercises, compared to previous courses, in which the impression was that students were less clear about what was happening when they carried out similar exercises.

6 | CONCLUSIONS AND FUTURE WORK

In this article we have set out the latest versions of ARMSim and QtARMSim, which provide a teaching simulated environment of the ARM Thumb architecture, multiplatform, open, free, and easy to use. These applications have been developed with the aim of enabling an easy understanding of how a processor works, and it has been widely agreed that both students and teachers feel that they have met this objective.

TABLE 2 Some of the answers to the question "What do you think needs to be done to improve QtARMSim?" obtained after the first year the simulator was used

Simplify the installation on Microsoft Windows.
The installation and the configuration should be more automatic.
Editor needs a little improvement, but above all, the installation process.
Its installation. It was quite difficult at the beginning to install it. Not only on Microsoft Windows, but on GNU/Linux. Some installation guides were later published, but it should be easier.
A brief explanation below each instruction on step by step execution o some animation. In fact, in the next Computer Architecture course, they use a program that provides an animation of how the instructions are executed on a MIPS processor.

As future work, we would like to provide new functions in the graphic interface as an animation of what happens when each machine instruction is executed, a simulated console, and a contextual help on machine instructions and data. In terms of the simulation engine, apart from providing the functionality that the graphic interface requires to accomplish some of the aforementioned desired new functions, we would like to add full support of ARM Thumb 2, expand the memory model to support simulated input/output devices, and add a cache simulator module.

References

1. Clements A. The undergraduate curriculum in computer architecture. *IEEE Micro* 2000; 20(3): 13–22.
2. Clements A. Selecting a processor for teaching computer architecture. *Microprocessors and Microsystems* 1999; 23(5): 281–290.
3. Clements A. ARMs for the poor: Selecting a processor for teaching computer architecture. In: IEEE; 2010: T3E 1–6.
4. Harris S, Harris D. *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann . 2015.
5. Barrachina Mir S, Fabregat Llueca G, Martí Avilés JV. Utilizando Arduino DUE en la docencia de la entrada/salida. In: Canaleta X, Climent A, Vicent L., eds. *XXI Jornadas sobre la Enseñanza Universitaria de la Informática*; 2015: 58–65.
6. Barrachina Mir S, Castillo Catalán M, Claver Iborra JM, Fernández Fernández JC. *Prácticas de introducción a la arquitectura de computadores con el simulador SPIM*. Pearson Educación . 2013.
7. Nikolic B, Radivojevic Z, Djordjevic J, Milutinovic V. A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization. *Education, IEEE Transactions on* 2009; 52(4): 449–458.
8. IEEE/ACM Joint Task Force on Computing Curricula. Computer Engineering. . Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering. tech. rep., IEEE Computer Society Press and ACM Press; 2004.
9. Clements A. *Computer Organization and Architecture. Themes and Variations*. Cengage Learning . 2014.
10. Hohl W, Hinds C. *ARM Assembly Language: Fundamentals and Techniques*. CRC Press. 2 ed. 2014.
11. Horspool RN, Lyons D, Serra M. Armsim#—a customizable simulator for exploring the arm architecture.. In: ; 2009: 223–228.
12. Barrachina Mir S, Fabregat Llueca G, Fernández Fernández JC, León Navarro G. ARMSim y QtARMSim: simulador de ARM para docencia. In: Canaleta X, Climent A, Vicent L., eds. *XXI Jornadas de Enseñanza Universitaria de la Informática*; 2015: 2–9.
13. Barrachina Mir S, Fabregat Llueca G, Fernández Fernández JC, León Navarro G. Utilizando ARMSim y QtARMSim para la docencia de Arquitectura de Computadores. *ReVisión* 2015; 8(3): 17–30.
14. Ardito C, De Marsico M, Lanzilotti R, et al. Usability of e-learning tools. In: ACM. ; 2004: 80–84.
15. ARM Limited . *ARMv7-M Architecture Reference Manual*. 2010.

AUTHOR BIOGRAPHY

Sergio Barrachina Mir. is an associate professor of the Computer Architecture and Technology Area at the Jaume I University (Spain). His research interests include high performance computing and architectures. He received his PhD in Computer Engineering from the Jaume I University.

Germán Fabregat Llueca. is an associate professor of the Computer Architecture and Technology Area at the Jaume I University (Spain). His research interests include fault-tolerant computing, embedded systems and networks of sensors. He received his PhD in Physics from the University of Valencia.

Juan Carlos Fernández Fernández. is an associate professor of the Computer Architecture and Technology Area at the at Jaume I University (Spain). His research interests include cloud systems and power-aware computing. He received his PhD in Computer Science from the Polytechnic University of Valencia.

Germán León Navarro. is an associate professor of the Computer Architecture and Technology Area at the at Jaume I University (Spain). His research interests include embedded systems and FPGAs. He received his PhD from the Jaume I University.

How to cite this article: Barrachina S., G. Fabregat, J.C. Fernández, and G. León (2020), QtARMSim: an ARM simulator for teaching Computer Architecture, *Journal of Software: Practice and Experience*, *2020;xxxx.*