Virtualization for Cost-effective Teaching of Assembly Language Programming

José O. Cadenas, *Member, IEEE*, R. Simon Sherratt, *Fellow, IEEE*, Des Howlett, *Senior Member, IEEE*, Chris G. Guy, *Senior Member, IEEE*, and Karsten Lundqvist, *Member, IEEE*

***Abstract*—A virtual system that emulates an ARM-based processor machine has been created to replace a traditional hardware-based system for teaching assembly language. The proposed virtual system integrates, in a single environment, all the development tools necessary to deliver introductory or advanced courses on modern assembly language programming. The virtual system runs a Linux operating system in either a graphical or console mode on a Windows or Linux host machine. No software licenses or extra hardware are required to use the virtual system, thus students are free to carry their own ARM emulator with them on a USB memory stick. Institutions adopting this, or a similar virtual system, can also benefit by reducing capital investment in hardware-based development kits and enable distance learning courses.**

***Index Terms*—Microprocessors, Virtual Machining, Software Libraries, Electronic Learning.**

I. Introduction

THIS document demonstrates how students' microprocessor laboratory experience and assignments can be improved using a virtual system built around the processor emulator QEMU [1] for the teaching and learning of assembly language for ARM processors. The proposed system uses a standard PC equipped with a native x86 processor running Windows or Linux (referred to as host) where an application is launched that runs Linux on an emulated ARM system (referred to as the guest). The virtual ARM system is logically indistinguishable from a physical board-based ARM processor system. The resources to run the guest virtual ARM system are made available to students as a compressed downloadable zip file [2]. When these resources are uncompressed, a folder with an executable and some data files is generated requiring around 1GB of storage space. Once the virtual system is running on a host machine, students have the option to interact with the guest machine using an X11 graphical interface [3] or in console mode (text only). When a student logs on to a guest account, all tools required for software development such as text editors, assembler, compiler, linker, debugger, etc., are natively available for targeting an ARM processor.

The virtual solution presented in this paper was adopted as a general replacement for the traditional hardware approach of using physical embedded boards equipped with a target processor, typically physical 8-bit processors, or 32-bit processors such as the 68K, MIPS and embedded PIC 8/16/32-bit processors [4]. This new virtual system also replaces the need for having cross-tools running on a Windows host (cross-compilers) thus the cost of software license fees is saved. In addition, there is no need for extra hardware to be purchased and/or replaced, such as proprietary tools for communication and debugging for the embedded board from the host machine. An immediate benefit of this approach is that a development environment can be replicated quite easily by students on their own PCs anywhere, anytime, since the whole system fits onto a standard USB memory stick. Assisted labs are still scheduled, mostly for students to get direct support from teaching staff. Students can dedicate any extra development time as a matter of personal choice without them requiring access to any extra university resources such as hardware/software or PC laboratories.

II. A Quick Review of Virtualization and Emulator Systems

Virtualization is a rich and wide concept; in fact, it has been applied in different forms at different levels for over forty years [5]; as a cost-effective way for software development [6], or for single servers to run different operating systems [7]. With tools such as VMWare [8] and VirtualBox [9], virtualization has now become easier and popular for all kind of users. Virtual machines, created with these two popular tools, currently cannot emulate an ARM processor and this explains the choice to use QEMU. QEMU can emulate several systems with different CPUs [1]. It has previously been used for mixed simulations of hardware and software models [10]. Also, dual-core virtual platforms to validate the functionality of hardware and software have been demonstrated [11].

An emulator is a way of allowing program code to run in a controlled environment with facilities that may, or may not, actually exist. The problem with emulators comes down to speed and resources such as storage and input/output devices. The emulator program has to execute several lines of software code in order to replicate the functionality of a totally different single machine instruction; this may only require a few nanoseconds in state-of-the-art hardware. In terms of storage,

Manuscript received XXX. This work was supported in part by the University of Reading teaching grant scheme 2011.

The authors are with the School of Systems Engineering, University of Reading, Berkshire, RG6 6AY UK (e-mail: o.cadenas@reading.ac.uk, sherratt@ieee.org, dphowlett@ieee.org, c.g.guy@reading.ac.uk, k.o.lundqvist@reading.ac.uk).

Digital Object Identifier

the computer running the emulator must have access to enough space for itself to run, as well as space for part or the whole of the emulated system. Also, since the emulator is a computer program, it can be adapted more easily than physical hardware.

The first emulators were concerned with the functionality of the much simpler systems than those on which they were run. Small 8-bit microcontrollers could be emulated on a PC at a fraction of real-world speed while providing the full range of debugging capabilities so that microcontroller code could be tested without having to build a circuit. A popular example of an emulator is the Multiple Arcade Machine Emulator (MAME) [12]. As host computers have become more powerful and capacious, it has been possible to emulate ever more complex systems. The use of emulators to support teaching concepts in computer architecture is not new. A popular teaching environment was SPIM, an emulator for the MIPS 3000 architecture [13]. SPIM was a large step forward in emulator technology but was not flexible enough for demonstrating the interaction between high-level and assembly languages. SimpleScalar [14] is still one of the most well known emulators. Although it is possible to teach assembly language using it, it is mainly used for performance analysis of applications based on specific processor features. But SimpleScalar does not run a full system; besides, it is common the need to use external cross-compilers when developing applications for it. Bochs runs a full system but does not emulate an ARM-based system [15]. Simics does emulate an ARM system but requires a license [16]. However, QEMU is a freely available emulator for a number of different processor architectures. For this work, QEMU running under Windows has been used to emulate an ARM processor of sufficient processing power that it can execute a complete Linux operating system.

Uses of virtualization for laboratory practices, in teaching and education, is recently reported frequently [17-19].

III. Building a QEMU ARM System

A QEMU-based system is composed of a number of software components. A simple way to understand the relationship across all these components is by having a look at the command and arguments used when invoking the emulated system to run. A run command has in general the following structure:

```
>qemu-system-arm  -M versatilepb \
–kernel Zimage –hda arm-lab.img –m 256
```

The command is actually passed in a single line ('\' is used to indicate that the line still continues). "qemu-system-arm" is the host executable to run a full system emulation for an ARM architecture ('>' indicates a command console prompt). This requires a QEMU installation on the host machine freely available from the QEMU web site resources [20]. The argument "versatilepb" is the specific ARM-based machine to emulate; one among some possible supported choices. The argument "Zimage" given after the "–kernel" flag is the name of the file generated after a Linux kernel compilation process. The argument "arm-lab.img" given after the "-hda" flag is the name of the file to be used as the hard disk for the emulated system. This corresponds to a Linux file system. The argument 256, given after the "-m" flag, requests the allocation of 256MB of memory space as main memory. A limit of 256MB is in place for the "versatilepb" machine.

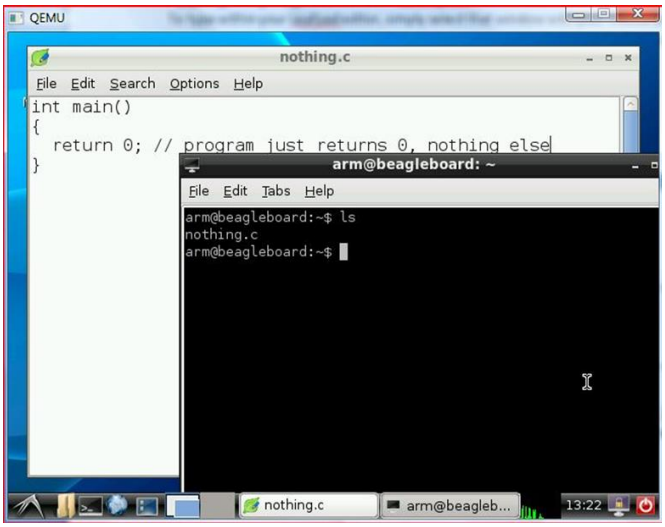

Fig. 1. The virtual QEMU-arm system running a graphical mode. A shell console is running as well as a text editor "LeafPad".

The success in getting a workable full system emulator is essentially due to a successful generation of two files Zimage and arm-lab.img.

A. Generating a Linux kernel for an ARM-based system

A Linux kernel is an operating system kernel released under the GNU General Public License [21] as C sources files. In order to run Linux on an ARM-based machine requires the generation of a Linux kernel executable targeted to an ARM machine. This process is well documented [21]; after a successful compilation process, the image will be ready as "zImage".

B. Generating a Linux file system for ARM

The quickest and possibly safer way to generate a root file system for ARM is to use automated scripts written specifically for that purpose. Firstly, a blank image file was created with a size of 1GB with the name "arm-lab.img". Secondly, this blank image was formatted with a Linux file system and then filled in with an uncompressed Ubuntu root file system as downloaded from scripts [22]. The generated root file system allows the configuration of a login user and password. Additionally, the lightweight X11 environment LXDE was also installed to allow for a graphical user interface [3].

IV. QEMU ARM System Interaction

Students are provided with all the resources required to run a full ARM system emulator. Versions for Windows and Linux host machines are available. A user's guide document is also delivered. The guide assumes no previous exposure to

Linux, however its structure allows for skipping sections for students already familiar with certain Linux topics. The major topics covered in the student's guide are explained in more detail below.

A. Booting up the System

A script is provided that when executed will pop up a window that will prompt to enter a login name identity. All students get access to the virtual machine with a login name as "arm" and a generic password. Students can opt to personalize their own login and password. First, users are guided to exit the system and regain access to it. Next, basic Linux commands are also introduced, such as, how to create files and folders and how to navigate across directories. Students also learn how to properly shutdown the system. Fig. 1 shows the system running in graphical mode.

B. Editing Text Files

To program in assembly language students need to edit text files. Therefore, students are introduced to use common text editors in Linux such as "vim". However, our experience is that some students may struggle to grasp this at first. This is where running the system in graphics mode is an advantage. Students learn how to switch from console mode to graphic mode and back. In graphic mode it becomes very intuitive to use the editor "Leafpad" embedded in the X11 LXDE desktop (Fig. 1). This editor is similar to the common "Notepad" in Windows. A shell Linux console as in Fig. 1 is where students get to write Linux commands. Yet a third method for editing is provided. All registered students are given a centralized network space referred to as the Network Drive (ND) by the University IT department. A script for students to map their private ND locally onto the running virtual ARM system is provided. As the user's ND is automatically mounted onto the Windows host where the virtual machine is running, then students are actually sharing their ND storage between their virtual Linux ARM system and the host PC where they are working. This has the benefit that students can opt to edit their Linux programming text files (to be used in the virtual machine) under their favorite text editor in Windows.

C. Compiling Projects

A progressive approach was adopted for students to get familiarity and confidence with the compilation process under Linux and the running of executables. Students start compiling (and finding errors in the compilation) very simple single-file C projects, and then move on to multiple-file C projects. Students learn standard procedures for building executables using the Linux utility "make", including linking of external libraries and how to include all these steps into the building process with tailor-made "make" files.

D. Preparations for Assembly Language

Students first experience the whole compilation process by invoking the compiler with a verbose flag on. This allows seeing the whole tool chain in action, in particular the role of the assembler and linker. Lecturers spend time with the students examining the structure of the generated assembly file from C programs. Students then learn how to use the assembler to generate object code, and finally use the linker to generate an executable. In particular, the table of symbols of an executable is examined aimed at discerning when and how to strip an executable to reduce its size. Students learn the difference between dynamic linked executables and static linked executables and how to find dynamic libraries dependencies from executables.

E. Writing Assembly Language

An assembly language file generated from a minimalistic C file program is used as a starting point. The assembly file is manually striped down to "bare.s" as the simplest file that could still be successfully executed under Linux. It is so simple that it has only five lines of text and 42 characters. At this stage many simple yet powerful actions are performed directly at assembly language level. These hands-on experience exercises a rich set of key module learning outcomes centered on the instruction set architecture (ISA) for manipulation of data and memory addressing modes.

Students are then focused on a small project entirely written from scratch in assembly language. Memory data alignment and the use of the standard method for procedure calls for a given processor architecture is learned. Specific use of processor registers for roles such as program counter, link register, stack pointer is discussed. The role of the frame pointer and their careful manipulation in a program before and after a function call is also learned. A key concept introduced here is the creation of stack frames or activation records [23].

F. Debugging

The GNU debugger (GDB) has been integrated for use within an assembly language program. Students learn how to compile for debugging, set break points, run a program in steps, examine registers such as the stack pointer and program counter, examine the program stack and walk through the stack frame operation. Students later elaborate on how to receive and manipulate global arguments passed to programs at the time of execution. Also, students practice how to pass and preserve multiple arguments to function calls.

G. Optimizing C Programs in Assembly Language

Although students create whole projects entirely written in assembly language it is more likely they will write projects in a high level language such as C. The opportunity here is to show how to take advantage of their knowledge of assembly language. Students are introduced to the analysis of their executables in order to derive metrics on selected sections of their code through the use of profiling by using the Linux utility "gprof" [24]. In particular, a simple program is used to compute the Fibonacci numbers using recursive calls. This illustrates, for instance, how to determine the number of calls performed by a running program to compute the Fibonacci number in a sequence. Students then have a detailed view at the assembly language level of the Fibonacci function and think of an optimization strategy that could be used to minimize the number of calls previously determined. Optimization is then applied by manipulating the function

code directly at assembly language level, re-compiled and run to check whether the unmodified and optimized versions are consistent and safe. For the specific optimization performed in labs (a lazy activation record method [25]), students see that the optimized version runs much faster.

H. Inline Assembly Language

There are instances where it makes sense to use assembly language embedded into the C code. Students are given specific examples of how to insert single or multiple lines of assembly code as well as the risks associated in doing so.

V. Module Assessment and Student Feedback

Computer Architecture is a 100 contact-hour compulsory module in the second year at the School of Systems Engineering. This is taken by students enrolled to the Computer Science and Electronic Engineering degree programs. It is composed of two-hours of lecturing per week plus two hours of laboratory per week to do the work described in this paper. Extra contact hours span across the autumn and summer term. Enrolled students had previous coursework in digital logic, C programming and computer applications. The module provides fundamental knowledge of commercially successful computer architectures with emphasis on their instruction set, organization and hardware/software interface. The following assessable learning outcomes are expected:

- Develop the ability to quantitatively evaluate computer performance by using benchmarks suites
- Develop skills to improve computer performance by using software techniques
- Develop assembly programming skills from laboratory practical sessions

The module is assessed by a written exam and the lab component contributes 30% towards their final mark for the module. The lab component observed common lab practices for teaching engineering undergraduates [26].

In 2010 the old lab assembly language approach was phased out and the new virtual system was rolled in. Since then, a specific survey has been made available to all students taking this lab, to be completed on a voluntary basis. Around 70% of students have completed the survey without any extra reward in order to do so. The survey is composed of 15 questions, from which the following five, are specifically addressing issues directly related to the learning outcomes of the module.

1. Have the ARM lab sessions enhanced your understanding of a processor's hardware organization?
2. Do you think the lab sessions have prepared you to write ARM assembly code?
3. Are you convinced that there are cases where assembly code can speed up the overall execution time when embedded into C code?
4. Have the ARM lab sessions given you the opportunity to experiment directly with the instruction set of the ARM processor?
5. Have the ARM lab sessions made it clearer the stages in the execution of different instructions, such as load/store, ALU and branch instructions

TABLE I
AVERAGE RESPONSES TO FIVE SPECIFIC QUESTIONS
A voluntary survey is applied on line. This survey was introduced since year 2010 along with the new system described here.
The survey was applied to N=237 students

Question	SA	A	NAD	D	SD
1	10.8	51.3	27.0	10.8	0.0
2	2.7	45.9	24.3	24.3	0.0
3	32.4	62.1	2.7	2.7	0.0
4	13.5	72.9	8.1	5.4	0.0
5	10.8	67.5	16.2	5.4	0.0

Average percentages of the responses gathered in the last three years using a Likert-scale are presented in Table I. The scale used is Strongly Agree (SA), Agree (A), Neither Agree nor Disagree (NAD), Disagree (D) or Strongly Disagree (SD).

The following remarks can be stated from the responses detailed in Table I:

- Students overwhelmingly feel that they are more directly exposed to the instruction set of a processor (Q4, Q5) and only around 10% of students disagree they have gained understanding of the internal hardware organization of a processor (Q1).
- No one has felt yet that assembly language cannot be of benefit to speed up C code (Q3). In fact, the majority of them have witnessed evidence of that in the labs. One extra question (of the remaining 10 in the survey) confirmed that optimizations at the assembly language level can lead to faster executions.
- However students still do not feel confident enough to write assembly code (Q2). Less than 10% thought that the software development process was not clear and only 5% thought the course was not useful.

The five questions above are not directly related to the virtual emulating system. These questions are there to evaluate whether the learning outcomes of the module are being reinforced by the lab; in other words, what students thought they had learnt. Having gained confidence the new lab is in line with the learning outcomes for the module then it makes sense to have a closer look at what students thought on the virtual system. Consequently, the remaining ten questions in the survey, besides helping to complement the five questions

TABLE II
FINAL MARKS IN LABS AFTER ASSESSMENT
Years 2011-2014 are for the new system. Number of students is N, standard deviation is SD and minimum and maximum mark is Min, Max.
Marks are scaled 1 to 100.

Academic Year	N	Mean	SD	Min	Max
2008-09	36	56.3	16.8	22	83
2009-10	45	56.5	16.4	17	81
2010-11	61	59.1	16.4	14	88
2011-12	76	72	25.1	33	93
2012-13	61	73	19.6	17	100
2013-14	50	83	13.5	40	100

above, were also seeking to get a feeling about issues such as: the use of real hardware vs emulation, working anywhere/anytime vs specific scheduled labs, small cost vs no-cost solutions to students, use of Linux vs another OS, and on-line computer-based assessment vs written reports. Broadly speaking, students feel real hardware is slightly preferable than emulation (5:4) but even a small cost for a hardware kit put the preference back to the virtual system due to having no-cost while providing the flexibility of working anywhere/anytime rather than in scheduled labs (3:1). In 2010 30% of students felt familiarizing with Linux was making the lab harder yet in 2013 this has dropped to under 15%. This is probably explained by the fact that students get to know well in advance (from word of mouth or by reading ahead from the module syllabus) that Linux will be used in our labs. Consistently over the five years, over 90% of students prefer to be assessed using on-line computer-based systems rather than written reports.

Table II shows the final marks achieved by students in their assessment in the last six years. The oldest three years correspond to the old lab content and the newer three years to the new assembly language lab.

Just a quick glance at the table reveals the new lab has produced a significant increase in the final assessment marks. Although apparently simple, the circumstances over these years had a more complex scenario. In the year 2009-10 on-line assessment was introduced and maintained ever since. In Table II, only in the year 2008-09 the assessment was based on written reports. So, it is not easy to decide what really has made the final marks to go up recently as it may be due to the change to computer-based on-line assessment or the introduction of the virtual system. A simple single factor ANOVA analysis to Table II with the groups 2008-11 and 2011-14 indicates that indeed the median between the old system and the new system are different. A T-test analysis between year 2010-11 to previous years indicates that there was no significant difference ($p>30\%$) in the marks distribution by the introduction of the computer-based on line assessment that year. By contrast, the same analysis between further years does indicate there is a very significant change ($p<1\%$) across the marks after the new virtual system was introduced that cannot be attribute to the on-line assessment.

The school has a long tradition of gathering, evaluating and incorporating changes based on student feedback collected after every module has been delivered. This is comprehensive and covers both lectures and practical work. This school-wide feedback is assessed by a board of studies set to that end. Their independent statistics gathered from this module indicates the module has become gradually easier, clearer and more interesting. As the feedback is gathered for the whole module (lectures and labs where many small changes have occurred in between) it can only be postulated that the lab component has contributed towards this positive change.

VI. Discussion

The use of an emulated virtual system for teaching assembly language has proved very useful at the University of Reading in terms of reducing capital expenditure and allowing greater flexibility for teaching. As time has moved on since its initial introduction, the emulator has been adapted to take account of newer features such as increasing the amount of memory the kernel has available and improved methods of offline storage. One primary reason for developing this virtual system was reducing the cost per student seat and as such we have never delivered this lab with a real ARM system. Nevertheless, we have received informal feedback from students that have executed the lab experiments in real ARM systems available in their mobile phones and tablets with no issues to report. It is expected the impact of teaching assembly language with real hardware would somehow be similar to teaching embedded systems programming with real hardware [27].

The method preserves a hands-on approach to gaining skills for software development whilst maintaining the focus on the understanding of key concepts. Not only does the system keep the emphasis on building familiarity with front-end concepts such as a processor instruction set (ISA), but it also simplifies the understanding of some back-end key concepts such as the application of specific binary interfaces or the notion of activation records. (Specific topics regarding physical interfaces to input/outputs devices such as GPIO, timers, and interrupts are delivered to students in a different teaching module that focuses on embedded microprocessors using real hardware with a PIC32 architecture).

This proposal made it natural to incorporate the standard GNU toolsets for development and also for debugging and profiling using GDB and "gprof". With these tools it becomes straightforward to demonstrate how manual optimizations introduced directly into the assembly language code can improve performance of applications in cases where such optimizations had been unable to be applied by the compilation process itself. Further examples of useful code with direct manipulation of assembly language have been brought into the lab exercises such as bit twiddling routines [28]. Since the deployment of the virtual system students have assimilated concepts and techniques much faster than before. This has made a real difference, with considerable improvement on previous experience in teaching assembly language, judging by the feedback gathered from students over the last few years. Students have been motivated by this new approach. For example, some students have reported that they have been able to port the emulator to Mac operating systems. One student wrote a Linux script to connect the emulator running in his home machine to the university Network Drive. This script was checked and approved by the university's IT department and has now been adopted by all students. Also, students have engaged in cross collaboration by opening discussions boards using the university virtual learning environment and have also contributed to a number of improvements and minor corrections to the teaching material and user manual.

A large benefit, to both the students and the university concerns equipment. If hardware boards were used, enough would have to be purchased in order to have one each, or one

per pair of students. In a way, this presents similar challenges to adopting hardware description languages to teach digital logic as opposed to the most traditional approach of using physical gates, flip-flops, etc. [29].

The use of freely-available software means that it can be installed on as many computers as necessary, without the need to spend extra money and, crucially, the students can download it for use on their own personal machines. Experimentation is no longer confined to a limited amount of equipment in a fixed-time session. A common request over the years from students has been their wish to take hardware development boards away from the laboratories for them to use in their own time in their home. The system presented in this paper fulfils this student request, without cost or risk to the university.

VII. Conclusion

Changing to an emulator-based assembly language teaching environment has saved money and increased flexibility for the School of Systems Engineering at the University of Reading by reducing capital investment in hardware-based development kits. Feedback gathered over the last few years from students that have taken the module show an increasing level of satisfaction with the new system. It has also allowed students to have more time to learn with the environment since they are able to have their own copy on their personal computers or carry it with them on a USB memory stick facilitating distance learning courses.

References

[1] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conf.*, Anaheim, CA, 2005, pp. *41-46*. [Online]. Available: http://wiki.qemu.org/Manual

[2] José O. Cadenas. (2014, Sept.) University of Reading, UK. [Online]. Available: www.dsd.reading.ac.uk/qemu-arm-0.11.0-lxde.zip

[3] LDXE. [Online]. Available: http://www.lxde.org

[4] C. G. Guy, "Teaching microprocessors at university," *IEE Electronics and Power*, vol. *28*, no. *3*, pp. 253-256, Mar., 1982.

[5] R. Figueiredo, P. A. Dinda, and J. Fortes, "Resource virtualization renaissance," *IEEE Computer*, vol. *38*, no. *5*, pp. *28-31*, May, 2005.

[6] S. Seetharaman and K. Murthy, "Test optimization using software virtualization," *IEEE Software*, vol. *23*, no. *5*, pp. *66-69*, Sept.-Oct., 2006.

[7] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *IEEE Computer*, vol. *38*, no. *5*, pp. *48-56*, May, 2005.

[8] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends," *IEEE Computer*, vol. *38*, no. *5*, pp. *39-47*, May, 2005.

[9] M. Mahjoub, A. Mdhaffar, R. B. Halima, and M. Jmaiel, "A comparative study of the current cloud computing technologies and offers," in *1st IEEE Int. Symp. on Network Cloud Computing and Applications*, Toulouse, France, 2011, pp. 131-134.

[10] M. Monton, J. Carrabina, and M. Burton, "Mixed simulation kernels for high performance virtual platforms," in *IEEE FDL Forum on Specification & Design Languages*, Sophia Antipolis, France, 2009, pp. 1-6.

[11] P. Cheng-Shiuan, C. Li-Chuan, K. Chih-Hung, and L. Bin-Da, "Dual-core virtual platform with QEMU and SystemC," in *IEEE Int. Symp. on Next-Generation Electronics*, Kaohsiung, Taiwan, 2010, pp. 69-72.

[12] MAME. [Online]. Available: www.mamedev.org

[13] J. Larus. (1990). SPIM S20: A MIPS R2000 Simulator. [Online]. Available: http://pages.cs.wisc.edu/~larus/spim.html

[14] D. Burger and T. A. Austin. (1997, Jun.). The SimpleScalar toolset, version 2.0. *ACM SIGARCH Computer Architecture News*. [Online]. *25(3)*, pp. 13–25. Available: http://www.simplescalar.com

[15] K. P. Lawton, "Bochs: A portable PC emulator for Unix/X," *Linux Journal*, vol. *1996*, no. *29es*, pp. *7*, Sept., 1996.

[16] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: a full system simulation platform," *IEEE Computer*, vol. *35*, no. *2*, pp. *50-58*, Feb., 2002.

[17] M. Anisetti, V. Bellandi, A. Colombo, M. Cremonini, E. Diamiani, F. Frati, J. T. Hounsou, and D. Rebeccani, "Learning computer networking on open paravirtual laboratories" *IEEE Trans. Educ.*, vol. *50*, no. *4*, pp. *302-311*, Nov., 2007.

[18] S. Ros, A. Robles-Gomez, R. Hernandez, A. C. Caminero, and R. Pastor, "Using virtualization and automatic evaluation: adapting network services management courses to the EHEA" *IEEE Trans. Educ.*, vol. *55*, no. *2*, pp. *196-202*, May, 2012.

[19] L. Xu, D. Huang, and W-T. Tsai, "Cloud-based virtual laboratory for network security education" *IEEE Trans. Educ.*, vol. *57*, no. *3*, pp. *145-150*, Aug., 2014.

[20] QEMU. [Online]. Available: http://wiki.qemu.org/Main_Page

[21] Linux Kernel. [Online]. Available: www.kernel.org

[22] Ubuntu. [Online]. Available: https://wiki.ubuntu.com/ARM/RootfsFromScratch

[23] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The hardware/software interface*. San Francisco, CA: Morgan Kaufmann, 2005.

[24] GNU gprof. [Online]. Available: http://sourceware.org/binutils/docs-2.22/gprof/index.html

[25] M. Satpathy, R. N. Mahapatrg, S. Choudhuri, and S. V. Chitnis, "High performance code generation through lazy activation records," in *7th Workshop on Interaction Between Compilers and Computer Architectures*, Anaheim, CA, 2003, pp. 37-47.

[26] L. D. Feisel and A. J. Rosa, "The role of the laboratory in undergraduate engineering education" *Wiley Jour. Eng. Educ.*, vol. *94*, no. *1*, pp. *121-130*, Jan., 2005.

[27] J. González, H. Pomares, M. Damas, P. García-Sánchez, M. Rodríguez-Alvarez, and J. M. Palomares, "The use of video-gaming devices as a motivation for learning embedded systems programming" *IEEE Trans. Educ.*, vol. *56*, no. *2*, pp. *199-207*, May, 2013.

[28] Bit Hacks. [Online]. Available: http://graphics.stanford.edu/~seander/bithacks.html

[29] E. Todorovich, J. A. Marone, and M. Vazquez, "Introducing programmable logic to undergraduate engineering students in a digital electronics course" *IEEE Trans. Educ.*, vol. *55*, no. *2*, pp. *255-262*, May, 2012.

José O. Cadenas received a Ph.D. from Reading University in 2002 in computer science where he currently is a lecturer. He works mainly on modeling and simulation of digital designs in RTL.

R. Simon Sherratt (M'97–SM'02–F'12) has been a Lecturer in Electronic Engineering at the University of Reading since 1996 where he is now a Professor of Consumer Electronics. Professor Sherratt was an IEEE Consumer Electronics Society Vice President ('08-9), AdCom member ('03-08, '10-15). He received the IEEE Chester Sall Memorial Award in 2006 and is now the Editor-in-Chief of the IEEE Transactions on Consumer Electronics.

Des Howlett (M'12–SM'12) received a B.Eng. degree in Electronic Engineering from the University of Reading, UK in 1990 and the Ph.D. in digital logic in 2012. He has worked for 20 years in electronics design and applications, mainly for microprocessor companies.

Chris G. Guy (M'82–SM'94) is a Professor of Electronic Engineering at Reading since 2008. Currently, Professor Guy's research interests are in wireless networking. He is a Chartered Engineer and a Fellow of the IET.

Karsten O. Lundqvist (M'12) received a B.Sc. and Ph.D. in Computer Science from University of Reading in 2005 and 2010 respectively where he is an International Teaching Fellow since 2011. His online tutorial on Android games programming has had over 50 thousand registered students.