Universidade Federal do Paraná

Departamento de Informática

Roberto A Hexsel

On the Transition from 'Programmer' to "Systems Programmer"

Relatório Técnico
RT_DINF 001/2018

Curitiba, PR
2018

On The Transition From 'Programmer' To "Systems Programmer"

Roberto A Hexsel
Universidade Federal do Paraná
Departamento de Informática
roberto@inf.ufpr.br

Abstract

When one considers alternatives for the teaching of low-level programming, a simulation model can be a better tool than a hardware processor because the model exposes more information about the program's execution. The extra information helps the beginner assembly programmer understand the interactions between their code and the (modeled) hardware features. For advanced students dealing with the intricacies of system software, such as exception handlers and synchronization, the information provided by a simulated processor is far more useful than what can be gleaned from the opaque black box which encases the processor.

As second-year students face the challenge of writing software at the hardware-software interface, there are several abstraction steps that have to be climbed more or less at the same time, such as matching code written in C and in assembly, and dealing with the complex timing of the hardware devices. The students suffer needlessly because they often perceive cycles of trial and error as a failure, rather than a ladder that can lead to the solution to the problem at hand. We provide some indications on what the instructors might do to help students in gaining confidence to try and fail and then try again.

1 Introduction

Our university offers an eight semester BSc in Computer Science. By the end of the second year, the students have taken three courses on hardware (digital systems and computer architecture), and two on "systems software" and operating systems. In the first year, they take two courses on programming, one in Pascal and one in C. They study assembly programming in the second and third semester, on two courses, namely Microprocessors, and Systems Software.

This paper is an attempt to describe some of the difficulties involved in teaching the Systems Software course, and helping the students to transition from 'programmer' to "systems programmer". The systems programmer works nearer to the hardware, and at this level, the debugging tools available are rather crude, as compared to those available to application programmers. Furthermore, the systems programmer has to deal with timing and concurrency, which are normally hidden from application programmers.

In my experience, teaching systems programming for computer scientists is best done with simulators rather than with hardware systems, albeit the simulators must provide realistic models of existing systems. We make use of detailed simulators, coded in VHDL [1], a hardware description language that can be used to write models at various levels of abstraction. "Functional models" are very abstract and with little or no timing information, whereas "register transfer models" (RTL) are detailed models with precise, cycle level, timing information.

The computer model we use in the laboratories and design projects is called cMIPS, which is an 'implementation' in VHDL of the classical five-stage pipeline design of the MIPS microprocessor [8]. This processor is described in Patterson & Hennessy's *Computer Organization & Design: The Hardware-Software Interface* [16], which has been the textbook of our Computer Architecture (CA) course for over two decades. This book is widely adopted and is considered by many as the ultimate introductory CA textbook.

The source code for the simulator is available at `https://github.com/rhexsel/cmips`. A description of the simulated system, and usage, can be found in the repository, at `docs/cMIPS.pdf`. The code for the simulator is distributed under the GNU GPL v.3.

The next section touches on some of the reasons for writing the cMIPS model. The simulator is presented next, we then discuss some of the issues encountered while programming the exposed hardware-software interface, along with our classroom experience with the cMIPS simulator.

2 Related work

A brief detour is needed to explain a couple of, shall we say, quirks. The 32 bit MIPS instruction set has two implementation artifacts that affect the timing of the branch and jump instructions. In order to minimize delays introduced by control dependencies on the pipeline, the instruction which follows branches and jumps is, by decree, always executed. If no useful instruction may fill the *branch delay slot*, then the slot must be filled with a **`nop`**. There is also a data dependency on load instructions. The instruction which follows a load cannot make immediate use of the fetched datum; it must stall for one cycle because of a *load delay slot*. If no useful instruction can be performed on the slot, then a **`nop`** must plug the gap between the producing instruction (the load) and the consuming instruction. We shall need this information shortly.

The author's first experience with a VHDL model of the MIPS instruction set was with *miniMIPS* [6]. This model lacks several of the opcodes, there is no Control Processor (CP0), and the branch prediction circuit is incorrect. Further development of that model did not seem promising.

As a tool for teaching processor design, a somewhat more interesting model is *MRStd* [3]. MRStd models a single cycle processor and is intended primarily as an instructional tool, with only a handful of opcodes implemented. The author employed this model in Computer Architecture courses when the students were asked to implement several new instructions in order to execute the recursive factorial on the model. Unfortunately, MRStd is only suitable for handwritten assembly code because the model does not support two artifacts of the MIPS instruction sets, the infamous branch- and load-delay slots (please see Section 4). Compiled code must be post processed to hide the delay slots, and to the beginner student this additional step raises some difficult questions which are an unwelcome distraction.

With this experience – Minimips and MRStd – it seemed a good idea to write a model from scratch, one that would run compiled code unmodified, and would seem familiar to students reading Patterson & Hennessy's book. The first requisite was met. As for the latter, the result is rather doubtful given the complexity of the instruction set and its modeling (and 'implementation') in VHDL.

There are other implementations of the MIPS 32 bit instruction set, *e.g.* Plasma, a three stage pipeline coded in Verilog [17], a superscalar version [4], and the Harris' design described in [7], which was developed independently from cMIPS. Implementations do the MIPS instruction set can be found at repositories such as `opencores.org`. As the author

was bent on thoroughly understanding both the modeling process, and the processor model itself, the writing of a model was in order.

3 Simulation model

The author taught Computer Architecture for over two decades, in courses that made use of Patterson & Hennessy's textbook [16], and there was always need of some instantiation of the MIPS processor for students to play with. Thus, some years ago, a model for the MIPS 32 bit processor was written, with two requirements: (i) it should resemble the design presented in the book as closely as possible; and (ii) run unmodified code compiled by GCC. Hence, it had to be a complete implementation of the MIPS32r2 instruction set, as specified on the architecture manuals published by MIPS Technologies [13, 14]. Unfortunately, completeness brought out the inherent complexities of a real-life instruction set, which somewhat marred the original intention of close resemblance to the processor described in the textbook.

The cMIPS simulator is a detailed, cycle accurate, model for the MIPS32r2 'classical' five stages pipeline – hence the 'c' on the name – plus abstract models for a few peripherals, which turn the simulator into a more or less complete 'computer'. A block diagram of the VHDL computer model is shown in Figure 1.

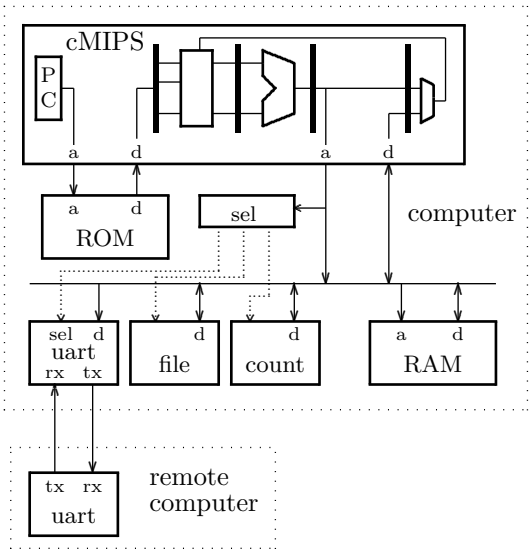

Figure 1: Block diagram of the "simple computer".

The cMIPS 'computer' is a complete and standalone tool for teaching CA, assembly programming at a more advanced level, and "systems programming", here understood as the writing of those components of an operating system which are closest to the hardware. The CA student can make use of the timing diagrams to follow single instructions down the pipeline. The assembly programmer benefits from observing all execution details of each and every instruction on a program. The systems programmer is exposed to the complete hardware-software interface, and the timing of all interactions between processor and peripherals.

The simulator models a "simple computer" with the processor core, RAM and ROM and a few 'peripherals'. The ROM memory is loaded with the binary resulting from compilation, and the RAM memory is loaded with initialized data. The 'file' peripheral supports the reading and writing of files, and the reading and writing of numbers and

characters through the host's standard input and output. The 'count' device is a 30 bit counter that can generate an interrupt after a programmable number of clock cycles. The 'uart' device is a serial communication interface (Universal Asynchronous Receiver Transmitter) that can communicate with a "remote computer" by making use of (arguably) the simplest communication protocol there is.

The design of a real-life processor core must include an extensive test suite to try to catch any new errors introduced by additions or improvements to the VHDL code. The test suite for the model includes some 120 assembly (>8000 lines) and C programs to test individual instructions and processor features, plus smallish C programs such as ten versions of integer sorting (>3500 lines of C code). A shell script automates regression testing. The "simple computer" is modeled in 10.000 lines of VHDL code. The code is distributed as free software so students can work on the processor model, and also with dozens of example programs.

4 On development and education

Here, the word *development* is meant to encompass two different activities. One is the development of the student's programming skills – to learn how to program a computer. In this context, the "programming of a computer" is done near the hardware-software interface, in assembly rather than Java. The other activity is the development of system software – how to program at the very interface between hardware and software, taking into account the hardware idiosyncrasies and the timing relationships between software requests and hardware responses.

Let's start with the easier one. Any implementation of the MIPS32 instruction set comes with two artifacts: branch delay slots and load delay slots. The instruction that follows a branch – the translation of an `if()` or a `goto` – is always executed; if there is no useful instruction in the instruction stream following a branch, a `nop` (no-operation) must be inserted after the branch by the programmer or compiler. Similarly for a load: if there is no instruction after a load which does useful work, a `nop` must be inserted after that instruction. A "useful instruction" makes the computation advance and does not break any control dependencies nor any data dependencies encoded in the instruction stream. The delay slots are an artifact of the early versions of the MIPS32 instruction sets, and were later removed.

These features ought to be hidden from the beginner assembly programmer, as the assembly language itself is enough of a hurdle. Thus, the initial steps in assembly programming are better taken with a simulator such as MARS [21] because a purely functional simulator provides the student with a simple and clean programming model. Hiding, or abstracting away, some of the hardware details from the beginner streamlines the learning process, as the novice programmer can concentrate on generating sequences of instructions while ignoring awkward hardware features.

More advanced assembly programmers must reckon with all of the hardware details, such as delay slots. MARS can be configured to expose these features to the programmer. My contention is that a less abstract model is the appropriate tool because it forces the initiated student to grapple with those not-so-tidy pipeline implementation artifacts. A detailed pipeline model is needed to close the semantic gap between "assembly instruction sequences" and the movement of those instructions through the pipeline stages. Once this is achieved, the more advanced assembly programmers are prepared to take on the development of system software.

The subtleties of interrupt handling and the implementation of synchronization primitives can only be fully exposed by a detailed model. Functional simulators do not expose,

at sufficient detail, the processor state, nor the state changes that occur during exceptional events. Timing diagrams derived from simulation can provide concrete instances of problematic timing and complex ordering of events. In the author's experience, the detailed model is more helpful than a hardware processor which, very efficiently, hides all internal activities from the programmer. The hardware-software interface must be *exposed*, rather then hidden, because this interface is the very 'thing' that is being programmed. In this context, the 'things' that are programmed may be the processor's control registers or some peripheral's control and data registers.

The idea of exposing the hardware-software interface becomes obvious when one is writing exception handlers: an exception such as overflow[1] has no – not a single one – external symptom to help the disgruntled programmer. Attempts to initialize a system with a virtual-physical address translation buffer (TLB) are bound to fall into the never ending exception loops caused by an empty page table. Without a simulator, these situations can be very difficult, or even impossible, to detect and fix.

The author strongly believes in learning by doing [9, 10] and having a simulator to work on helps enormously with the 'doing' part. Recent experiences in debugging the virtual memory subsystem convinced me that detailed simulation – at the instruction execution level – is the most comprehensive software debugging tool there is, perhaps a tool that shows rather more than one may want to look at. A detailed simulator may not be easy to use, but it can show each and every bit 'implemented' on the processor's model.

4.1 An early start

My first degree is in Electrical Engineering, and in the late 1970s, computer programming was something a tad esoteric. I was baffled when learned that the Fortran expression "`a = b + c`" bore little relation to the well-known mathematical expression "$a = b + c$".

From a pedagogical point of view, I believe the teaching of programming in an imperative language, such as C, Pascal or Fortran, which ultimately are simulations of a von Neumann Machine, must be accompanied by a simultaneous, clear and honest explanation of what the underlying machine is [9]. I do not claim to be original on this point, as Yale Patt, who is some 20 years my senior, has expanded on this idea thoroughly in his textbook [15].

The abstract machine presented to the student by a language such as C or Pascal embeds the idea of *state*, whereas all the mathematics the student has seen prior to the programming course have no state at all. Hence, the confusion between the stateless "$a = b + c$" and the state-changing "`a = b + c`" should be dispelled as early as possible.

To that effect, I believe that the first course on abstract programming (C, Pascal) must run in parallel with a course on concrete programming (assembly), where the underlying machine is clearly presented to the students. The concrete machine may be a highly simplified processor, such as Patt's LC-3 [15], or as a real but simple instruction set, such as MIPS32 [12, 13, 20].

If (i) the students are shown a somewhat realistic model of the concrete machine on which their C programs execute, and (ii) they understand the notion of a register, and (iii) what is a memory address, then it is easier for them to understand the relationship between the abstract C code and the concrete machine on top of which it executes.

From a systems programming point of view, this concrete machine can be used to impart, from the very beginning, the notion of time. The execution of a program takes so

[1] The addition of two operands may yield an incorrect small value because the number of bits needed to represent the correct result is larger than the word size. This programming error may go undetected if an exception is not raised by the processor.

many instructions, each instruction needs so many clock cycles to complete. The notion that execution takes time, that events inside the computer are not instantaneous, should be acquired early on. As trivial as this idea may seem, recall that our personal computers, whether desktops or mobile devices, respond almost always in a small fraction of a second. The general perception is that our computing devices provide "instantaneous answers" – of course, we computer scientists and engineers do know best, and we should inform our students that light travels 20cm per nanosecond on a piece of wire, and also what are the consequences of bits traveling at non-zero speeds.

5 Classroom experience

Our students are presented with spartan development tools: the `ghdl` VHDL compiler [5] and `gtkwave` to display the timing diagrams that result from the simulations [2]. C and assembly programs are compiled/assembled with `gcc` and `binutils` [18, 19]. Figure 2 shows `gtkwave`'s screen from a simulation run. The rectangles were added to highlight the progress of one instruction through the pipeline stages.

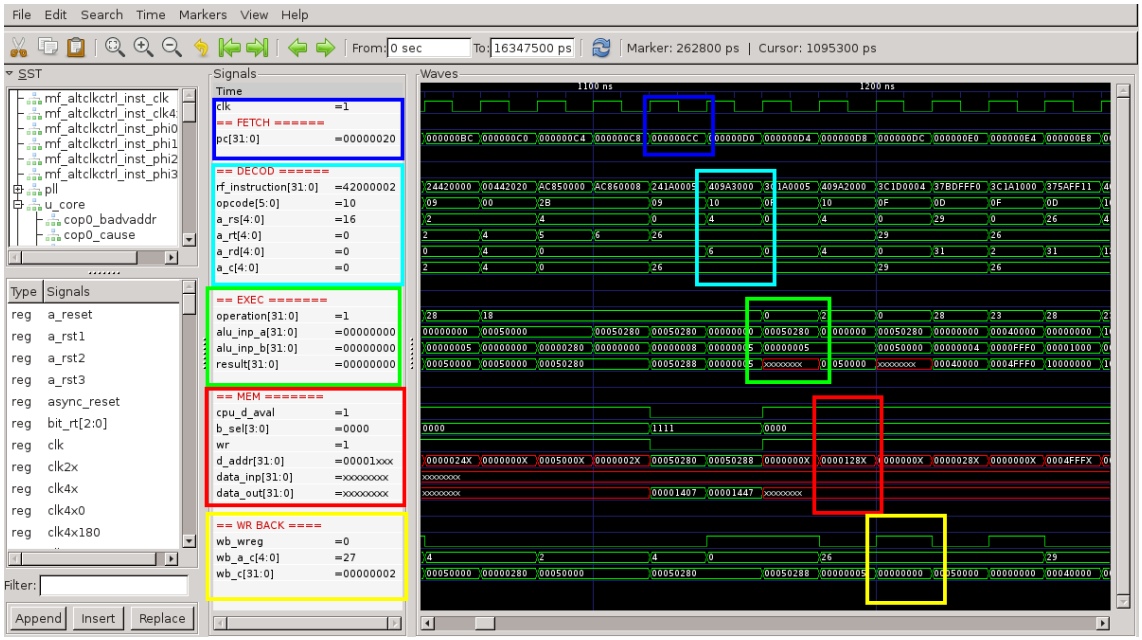

Figure 2: Sample simulation run displayed with `gtkwave`.

5.1 Third semester course

Over the last five years the cMIPS simulator has been put to the test in a 60 hour course, taken on the third semester. The students make use of the simulator in three laboratory sessions: (i) the first introduces the simulator and describes, in a step-by-step manner, its usage and features; (ii) then the students have to modify a small portion of an interrupt handler to implement a real-time clock that makes use of the 'count' peripheral; and (iii) the students are given an incomplete serial line interface driver and have to test and verify the interrupt handling routines, working only on the receiver-side. In the course's design project, pairs of students must implement an interrupt-based full-duplex communication protocol.

In the second and third lab sessions they have a first contact with matching a C program to an interrupt handler, written in assembly. To complete the project, the students must climb over a few abstraction steps. They have done some assembly programming and some C programming, and are now asked write two parts of one program at the two abstraction levels. This, in itself, does not seem to be much of a hurdle.

The remote computer sends data at a fast pace, and if any datum is missed, the program produces wrong results. Hence, the reception code must be lean and efficient, which is a requirement the students have not met before. The amount of data sent to the remote computer is about ten times that what is received, thus receiving and transmitting queues have to be managed in a timely fashion. This is where it all comes together: matching the actions of the interrupt handler, written in assembly, with those of the C program, under the constraints imposed by the dynamics of the communication protocols. This is a great deal of information to be managed at once, and superficial understanding of the concepts involved will not do.

The timing diagrams show hundreds of state bits. One of the difficulties is to choose what information is relevant. Additionally, relevance changes with the situation as some signals (wires or register contents) are relevant when all is well, but a different set of signals becomes relevant when something goes wrong. There are no simple rules because the sets of relevant signals are somewhat different for the code written by each team of students.

The solution written by the author is roughly four pages long, with two pages written in C, and two pages in assembly. In this case, size has little relation to complexity as timing issues and corner cases – queues that become empty and/or full – make the problem inherently complex.

A large proportion of the students start the programming part of the project by doing what one may call "programming by trial and error": (a) hurriedly write 100 lines of code; (b) compile and run said code; (c) find out it does not meet the specification; (d) add/remove/change some randomly chosen lines; (e) go to step (b). The tendency of programming by trial and error is displayed by many of the participating students, regardless of admonitions to do the contrary. Irony aside, my non-system colleagues agree with the perception that our beginner students apply this "programming by trial and error" more often than desirable. A similar "programming algorithm" is described by Lister *et alli* in [11].

In spite of repeated preaching that "programming is something to be done with pencil on paper", the students prefer "programming on the keyboard". My impression is that doing something with their hands feels to be much easier than solving the problem with their minds. They act as if their response time in solving the problem must be nearly instantaneous. The very urgency to complete the assignment hinders their progress.

People normally do experience a great deal of anxiety when dealing with something new and challenging. In this case, the students start up with the impression "they know everything needed" to complete the project. The first attempt ends in utter failure, they try again and fail. Then repeat. Most of them do not recognize these cycles as an important part of the learning process. Their understanding on the second cycle is better than on the first, and on the third better than on the second. After one cycle of trial and error, we never go back to a clean slate, as the errors are important to help us construct an understanding of the problem.

A large part of the anxiety stems from the false perception of "repeating and not making any progress". The process may be slow, it surely is frustrating, but there is always progress in an honest attempt.

From this point of view, the students have to deal with a meta-problem: they have to learn to cope with *not knowing*. It takes training for one to develop the faith that

learning will come in time, that learning takes time. A child starts with single letters, then syllables, simple words, sentences, and only four years later comes the gratification of being able to read Harry Potter.

Teachers face a different meta-problem. We have to create situation(s) in which the students must face the difficulties involved in patiently learning from mistakes. We also have to make sure that any alternative to not facing the meta-problem, not engaging in the learning process, is most unattractive.

A similar analogy is that of riding a bicycle: it cannot be taught, it has to be learned, through many falls and a few grazing of knees. Teachers have to tell students that they will not ride their intellectual bikes on the first day. We must tell of our own mental processes while developing the solution, we must tell of how many cycles it took us, so our experiences may lessen their sense of urgency.

This is not a simple undertaking. When told that it took me three days to complete the assignment – with several cycles of trial and error – rather than taking this information as friendly advice, meaning that the assignment is hard but do-able, some of the students took it as some kind of threat. It felt as if I was placed, by some students, in the rather uncomfortable position of omniscience, and when my limitations were exposed, this simple and honest piece of information was deemed unacceptable.

We need to provide the environment in which this process of learning to cope with *not knowing* may take place. The students have to be challenged while at the same time they must be made aware that we (teachers) do know the challenge is difficult. We must inform of the methods that may help, and that errors are an important part of the learning process. The idea that errors help us to develop our perception of the problem takes time to sink in, specially in people who grew up in the age of "instant gratification" provided by the Internet and mobile devices.

It should be stressed that "learning from trial and error" is different from "programming by trial and error". The first is a difficult and frustrating process that will be of use throughout their lives – it takes nine months for a child to be born. The latter is just a frustrating process that derives from their misplaced sense of urgency – two women cannot make a baby in four and a half months.

At the time of this writing, the course as described in the previous paragraphs has been through five cycles of trial and error. The specification of the project has been revised and improved at each instance, with more constraints added to make the problem less open ended. The samples are small from a statistical viewpoint, yet in later groups more pairs have successfully completed the assignment. The instructor is also learning from his mistakes.

5.2 Sixth semester course, elective

A different approach was attempted on two instances of an elective 60 hours course, with groups of 10 to 15 students, most of whom were nearing graduation. All the students had completed, or were currently enrolled in the Operating Systems (OS) course.

The students had to work through the same three laboratories as the beginners, then were asked to implement a producer-consumer pair, with the producer being an interrupt handler, and the consumer being an ordinary process, running with no OS infrastructure. The obvious solution makes use of two semaphores, one for mutual exclusion, and one for managing queue space. Much to the author's dismay, on each instance, only one of the pairs of students "did the right thing", which to my mind was to look up the solution in the OS book and adapt and adjust it to satisfy the specification. This involved mastering the two MIPS instructions for atomic synchronization – load-linked and store-conditional –

then writing the code for the semaphores in assembly, with less than a dozen instructions for each of the `P()` and `V()` primitives. The remaining pairings got mired in programming by trial and error.

Most of the students in these groups have not undergone the experience of "learning by trial and error" as described ins Section 5.1. It seems that the availability of development tools is not, by itself, conducive to learning. Programming by trial and error seems to be a far more appealing activity than climbing the learning curve of a tool that does help to solve the proposed problem. It also seems hard to make use of the intellectual tools, in this case semaphores, presented to the students in other courses. Often, students do not "take possession" of the subjects studied earlier. In my twenty odd years of teaching experience, knowledge appropriation seems to be a slow and erratic process.

From talking to the students during the laboratory sessions, I get the impression they are willing to spend a great deal of effort in learning how to use one of the fancy integrated development environment (IDE), such as Eclipse, but do not seem willing to learn what is deemed to be a low-level development tool, even if this tool is the only tool available for solving the problem at hand.

To the students, there seems to be no point in spending the energy in learning to use a tool that is only needed on "this elective course". The meta-knowledge to be gained in the process is ignored or disregarded: (i) how to debug at the hardware-software interface, when one has to keep tabs on a dozen registers and buses, each holding only a few (interesting) bits of state of the computation; (ii) how to record (long-ish) sequences of events which hide/expose an error; (iii) how to generate test vectors to pry out the incorrect behaviors, or to guarantee the system is performing correctly. All these activities demand a great deal of patience from the programmer.

6 Rantings (or In Conclusion)

The only excuse I can make for these pessimistic conclusions is that most of the students in the later discipline had not been exposed, early enough, to the contents of the introductory discipline. The more advanced students never had to deal with problems arising from 'real-life' timing constraints, and therefore did not have the necessary experience(s) – painful or otherwise – on this rather vexing subject.

My guess is that current computers are so fast that anything that happens inside them *must* be instantaneous; therefore timing problems do not have any bearing on the daily experience of younger computer users. Old engineers like the author had to put up with slow machines, until *circa* 2000, before computing became 'instantaneous'. Confined electromagnetic waves travel some 20cm in a nanosecond, which is anything but instantaneous. We must make sure our students understand the consequences of this harsh reality.

Part of the difficulty has to do with they not appropriating themselves of the tools, so those may be used to solve the problem(s) at hand. This 'appropriation' takes time and effort, something not within easy grasp to most people. Oddly, instead of making use of the information provided by the tools, many students come to regard "systems programming" as something rather frustrating. Indeed.

These issues, to a great extent, have to do with us, instructors. In a manner similar to the student's, the instructor has to learn how to make use of the tools, so that those will actually help the students. The issues seem to be: (i) do the students understand the questions posed? (ii) do they understand how the tools can help with the solutions? (iii) are the questions, the tools, and the expected solutions appropriately matched? In my experience, it takes about three attempts/semesters to learn how to match the problem to those charged with solving it. Initially too hard, then too easy, then about right.

All contemptible remarks in the previous paragraphs result from (hopefully) honest self-criticism, and from observing a relatively small sample of students, about 180 students on the introductory course over five semesters, and some 30 students in the advanced course on two semesters.

We return to the earlier point on "teaching patience". This issue is not widely perceived as problematic by many instructors, perhaps because few of us assign programming projects complex enough for the issue to become apparent. Furthermore, the undertaking is far too great for individual efforts; the whole CS Department ought to be involved if any significant gains are to be achieved. Finally, "patience learning" is itself a slow process; the student has to be presented with problems that generate increasing doses of frustration, concurrently with learning to endure frustration. As most parents would testify, this is easier said than done.

References

[1] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 3rd edition, 2008. ISBN 978-0-12-088785-9.

[2] A. Bybell. GTKWave – visualization tool for VCD, LXT, LXT2, VZT, FST, and GHW files, 2012. <http://gtkwave.sourceforge.net>, 11/7/2012.

[3] N. Calazans and F. Moraes. Simulação VHDL do processador MRStd. Technical report, Faculdade de Informática, PUC-RS, Brasil, 2006.

[4] R. Carli. Flexible MIPS soft processor architecture. Msc dissertation, Massachusetts Institute of Technology, Dept of Electrical Engineering and Computer Science, 2008.

[5] T. Gingold. GHDL – G Hardware Design Language, 2012. <http://ghdl.free.fr>, 11/7/2012.

[6] S. Hangouët, S. Jan, L.-M. Mouton, and O. Schneider. miniMIPS. Technical report, Opencores.org, 2009.

[7] D. Harris and S. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2nd edition, 2012.

[8] R. A. Hexsel. cMIPS – a synthesizable VHDL model for the classical five stage pipeline. Relatório Técnico RT-DINF 001/2016, Depto. de Informática, UFPR, 2016. <http://www.inf.ufpr.br/roberto/rt001_2016.pdf>.

[9] R. A. Hexsel and R. Carmo. Ensino de Arquitetura de Computadores com enfoque na interface *Hardware/Software*. In *WEAC'06: Workshop sobre Educação em Arquitetura de Computadores*, pages 9–16, 2006.

[10] R. A. Hexsel and R. Carmo. cMIPS – uma ferramenta pedagógica para o estudo de arquitetura. In *WEAC'13: Workshop sobre Educação em Arquitetura de Computadores*, pages 1–4, 2013.

[11] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad. Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *SIGCSE Bull.*, 38(3):118–122, Jun 2006.

[12] MIPS. *MIPS32 Architecture for Programmers, Volume I: Introduction to the MIPS32 Architecture*, 2005.

[13] MIPS. *MIPS32 Architecture for Programmers, Volume II: The MIPS32 Instruction Set*, 2005.

[14] MIPS. *MIPS32 Architecture for Programmers, Volume III: The MIPS32 Privileged Resource Architecture*, 2005.

[15] Y. N. Patt and S. J. Patel. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. McGraw-Hill, 2nd edition, 2003.

[16] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 5th edition, 2014.

[17] S. Rhoads. Plasma - most MIPS I opcodes. Technical report, Opencores.org, 2001.

[18] R. Stallman et al. GNU Binutils. Technical report, Free Software Foundation, 1998. <http://ftp.gnu.org/gnu/binutils>, 28/7/2013.

[19] R. Stallman et al. GCC, the GNU Compiler Collection. Technical report, Free Software Foundation, 1999. <http://gcc.gnu.org>, 28/7/2013.

[20] D. Sweetman. *See MIPS Run – Linux*. Morgan Kaufmann, 2nd edition, 2007. ISBN 0120884216.

[21] K. Vollmar and P. Sanderson. MARS: An education-oriented MIPS assembly language simulator. In *ACM SIGCSE Bulletin*, pages 239–243, Mar 2006.