Improving Instruction Set Architecture Learning Results

José M. Claver, María I. Castillo, Rafael Mayo
Dept. of Computer Science and Engineering
University Jaume I
12080 Castellón (Spain)
E-mail: {claver|castillo|mayo}@uji.es

Abstract

In this article, we put forward a new methodology and strategy for teaching the Instruction Set Architecture in a "Computer Organization" unit. This unit belongs to the second year of the undergraduate program in Computer Science at our University. In particular, we have centered our effort on the development of laboratory sessions, focused on an adequate introduction of assembler language of a general purpose processor. Our methodology has taken into account, among other aspects, the choosing of processor and tools, the pace with which concepts are introduced and the responsibility of learning. Thus, we obtain a less traumatic approach for our students to one of the most important subjects in the background of our future computer engineers and scientists.

Results obtained show students response is positive. This effect is reflected in student's interest and the ease with which they are been able to solve the exercises we set them to do. Because of that, an improvement on learning in this subject and this aspect is reflected in the subsequent evaluation of students.

1. Introduction

Generally, Computer Architecture units taught in Computer and Computer Engineering undergraduate programs begin during the first year, and are followed in subsequent years, with varying intensity and depth depending specialization. The contents of these units include many aspects that are fundamental for the training of future computer scientists and engineers. It is lively that these aspects will retain their importance in the syllabus for the next 10 to 20 years [2], since the development of applications and their performance requirement have an important impact on the architecture, structure and organization of computers [9, 10, 12]. From another point of view, we do not foresee over the next years any important modifications to the way computers work and are constructed. Nowadays, in the architecture of current processors we are using some ideas that appeared more than 30 years ago. However, we are expecting new developments that will make possible very different approaches to the widely used computational models, like quantum and molecular computing [3].

There is a generalized consensus about the contents that must be taught in the first computer architecture courses [1, 2], but there are different alternatives with respect to the kind of processors and tools used for describing and studying how they work [5]. In this sense, it is particularly interesting to study processor organization, instruction set architecture (mainly assembler programming), memory use and input/output control. These contents are, for this reason, the core of first units on Computer Architecture. Nevertheless, the high sophistication and configuration variety of current computers means that these must be studied in varying depth according to the final course goals [6]. An important complement of computer architecture contents is the study of the technological aspects of computer design, but we do not consider these aspects in our article.

As we will see in this paper, the choice of processor and the tools used to teach this subject are very important, but the pace of the course and responsibility have also a decisive influence on the learning of the concepts and techniques we want our students to learn. These two elements

determine whether or not students can achieve the planned goals. If care is not taken to adequately adjust the pace of the course and encourage a responsible attitude on the part of the students, many of them will lose the thread of studies and finally drop out.

The rest of this paper is organized as follows. In section 2, we review some more outstanding aspects related to teaching in introductory Computer Architecture subjects. In section 3, we compare the experience of teaching this subject at other Universities. In section 4, we show the more important aspects of our teaching methodology. In the last section, we explain the principal conclusions derived from our experience.

2. Introductory teaching in Computer Architecture

Introductory Computer Architecture units include, among other digital aspects, the study of following lecture topics:

- Processor: Structure and organization, data path, instruction set and machine language, assembler language and their relation to high level languages.
- Memory: Organization and management, instructions and data storing, and cache.
- Input/Output: Asynchronous control by status register and exceptions, and protocols.
- Performance: Analysis and comparatives.

These contents must be transmitted to students both in theoretical lectures in the classroom and in laboratory work. In this way, they can acquire the knowledge and abilities set out in the unit syllabus.

The most important conceptual aspects must be given a firm basis in theoretical lectures and then extended and elaborated in laboratory work where the conceptual elements are given practical application. In this context, the pace and the order of laboratory work must be adjusted to theoretical lectures. Furthermore, in laboratory work students acquire the ability to perform analysis and synthesis. For that, it is often to enhance assembler language study (exoarchitecture) by using a real machine or simulator of an architecture strongly related to the model explained in previous theoretical lectures (associated to endoarchitecture and microarchitecture [7]), preferably the same.

There is a time in the design of an academic project in which we must adopt, among various alternatives, an example processor and a tool to teach this subject.

2.1. Processor

It seems clear that it is preferable to choose a real processor rather than a hypothetical one as this will mean that it can be used as a part of a real device such a commercial computer or a development system designed for laboratory. It is impossible to do this with a hypothetical processor. Nevertheless, real processors introduce some special features that cannot be extended to other processors, since its design is the result of practical and economic considerations. These special features can introduce, during the first years, additional complexities in the learning process. Furthermore, the perspective acquired by students may be not adjusted to more extended processor architecture. So, it is important to choose an suitable processor.

On the other hand, hypothetical processors can be better adapted to academic needs at each moment, in function of student knowledge level. Thus, it is easier for them to assimilate and apply the concepts and techniques involved. These processors are often designed by choosing different abstraction levels of some more extended general-purpose processors.

2.2. Tools

The tool chosen to teach this subject may be a commercial computer, a development system, or a simulator. A commercial computer can be directly used if a real processor is chosen. In this case, the assembler and software development used must be adapted to the characteristics of a particular processor, as well as its possibilities and restrictions. In this situation, processor analysis is indirect and limited by the organization of a particular machine (memory, cache, bus, etc.).

When a simulator is used, it can be designed for a real or hypothetical processor (original or as abstraction of a real processor). Simulators of hypothetical processors could be better adapted to

the needs of a particular unit. Furthermore, complexity can be increased in subsequent courses by extending and modifying initial abstractions. Thus, it is possible to introduce more advanced concepts one at a time without students needing to learn about new tools or assembler languages.

In both the above cases, it is possible to use computer resources (directly or no), although these resources are always more limited if we use simulators.

3. Experience in other Universities

In recent years, teaching experience on introductory computer architecture subjects has featured by following characteristics:

- Choose, for the first year, a simple hypothetical processor instead of a real 8 bit processor (i8085, Z80, MC6800, R6500, etc.). The main goal is to reduce the gap between student knowledge and introductory concepts in the first years of computer architecture.
- In the second year, it is typical to opt for one of the well-known real 16 bit CISC processors produced in the 80's (Intel i8086, Motorola MC68000, etc.). In some cases, a 32 bit RISC processor is selected (MIPS R2000, ARM, MC88010, etc.), in order to reach a better approach to the complexity of current commercial processors [5].
- Simulators, alone or combined with development systems and market processors, are used to show the relationship between assembly/machine languages and architecture, to appreciate the challenge of producing efficient and correct programs, and to develop applications with real hardware.

4. Our proposal

In the first year of our Computer Science program, the "Introduction to Computers" unit presents elementary concepts about how computers work, and the basic switching logic. In the second year, "Computer Organization" course presents a more detailed study of computer architecture and organization. Important parts of this unit are the instruction set architecture and the assembler programming.

Until last year, the processor we were using (since 1992) to teach "Computer Organization" was the Motorola MC68000, one of the most elegant exponents of CISC architecture. There are a lot of reasons why MC68000 is the most used processor to teach Computer Architecture (its streamlined architecture combined a powerful instruction set with moderately easy-to-learn assembly language). Nevertheless, its architecture is far from the seminal processors of current superscalar architectures.

In laboratory classes, students used a MC68000 based development system with a tedious environment which cannot be used outside laboratory. Each laboratory session was organized as follows: it began with a lecture by the teacher, in which many new concepts were presented and the goal of each session was fixed. These goals were well specified, but very ambitious, and close attention by students was required. Furthermore, before students began to work, they need a meticulous study of its paper description, in which was included an abstract of the principal concepts explained by teacher. These two tasks take up the greater part of the laboratory sessions. Because of this, students had little time to develop the proposed exercises.

Laboratory sessions were designed in increasing order of complexity, as students were to analyse and design assembler programs, of variable complexity, from the first session. These programs included data declaration, different kinds of instructions, and a great variety of addressing modes. This situation is habitual in the assembler language teaching of analyzed universities.

All the above circumstances encourage us to plan an alternative in order to make it easier and more comfortable for our students to reach the goals we set them in these laboratory sessions. We base this change on the following initial goals:

1. Simple processor architecture.
2. Abstraction of a more advanced real processor.
3. Simulator easy to use.
4. Laboratory work can be continued at home.
5. Self-learning.
6. No need to attend with fixed timetable.

7. Personalized rate of learning (asynchronous learning)

In order to obtain the two first conditions we opted for a non-segmented abstraction of the MIPS R2000 processor, which has easier structure and is easier to program than a real R2000 processor [12]. This is a RISC processor; therefore, it has a simple instruction set and reduced addressing modes. Furthermore, in more advanced units we can use the same or similar processor (as the hypothetical, but realistic, DLX processor), without current abstractions [8, 10].

Conditions 3 and 4 are obtained by using the SPIM simulator. Concretely its graphic version called XSPIM, developed by James R. Larus from the Wisconsin University, which works under Linux and DOS/Windows operating systems [11]. This is an integrated simulator, where all information about processor and memory can be shown, and it is easy to use. The latest version (from the 6.3) of this simulator can show original R2000 programming difficulties, activating delayed load and delayed branch functionalities. Furthermore, as this simulator is a freeware software and multiplatform, students can use it at home.

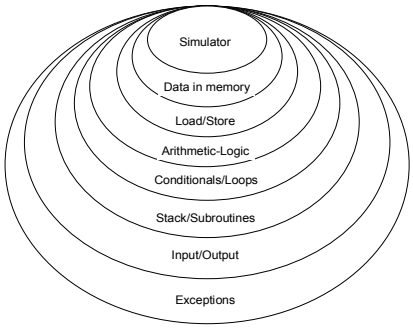

Figure 1. Contents of corresponding laboratory sessions.

The last three conditions are reached thanks to the planning and development of an adequate laboratory textbook. Each part of this textbook is associated with one or several laboratory sessions and is self-contained. In each chapter new concepts are presented, the more advanced the unit, the more complex they become and it is supposed that only previous session concepts are known. Figure 1 shows a concentric vision of contents corresponding to laboratory sessions.

We have taken special care with students learning rate when designing the contents of each session. We don't forget the maxim that says: first analyze and after synthesize. For that reason, all sessions begin with a little introduction and several example programs that students must analyze and understand the behavior of. In the following step, we propose some changes to previous example programs that students must analyze at another time. Thus, students increase their participation and make some simple guided synthesis. Finally, synthesis problems, such as short development projects, are proposed in order to test the correct understanding of the techniques and concepts introduced. In the next chapter/session, students begin with other example programs that they must analyze, and follow the same steps again. These steps are shown in Figure 2.

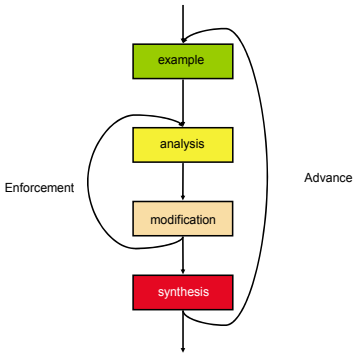

Figure 2. Learning flow of laboratory sessions.

Thus, laboratory work is structured as a consecutive set of questions which require from students: to analyze an example of assembler program, to modify them, and complete a design based on the concepts and techniques they have learned (see Figure 3 for an example of this structure work). A textbook containing laboratory work is also electronically obtainable from the unit website [4].

When students have doubts about some concept or technique, they only have to review earlier sessions. Thus, there is no need for a teacher to be near the student at all times, and

students work, systematically, solving questions and learning actively, because there is no obligatory attendance at laboratory sessions. Logically, students work at their own learning pace. Students needing more time on a particular session know that they have to work outside laboratory programmed sessions, either in free time access or at home, to finish all laboratory sessions. Only in this way, do students have any guarantee of passing the evaluation of assembler programming.

Example	Made a file with the following code: … Description: …
Analysis	**Question 1.** What code does …? ¿What is the value of...? **Question 2.** Show which instructions do...? **Question 3**. If datum ... has a 5 what happens...?
Modification	**Question 4.** Modify the code to ....
Synthesis	**Question 5.** Implement a program ....

Figure 3. General structure of laboratory work.

Since simulator use is not an end goal of this laboratory, students don't have to show their ability to use it in the evaluation tests. Otherwise, in the laboratory they analyze, modify and design little programs by using concepts and techniques shown in theoretical lectures, and these subjects are the core of evaluation tests.

Figure 4 shows the number of students who have passed (results great than or equal to 5) and the overall of results obtained at course 00/01, in which the new methodology is introduced, have increased respect to early years (98/99 and 99/00). Tendency lines of last two years highlight this behaviour.

The number of students attending exams rose 20%, as dropouts are reduced (these are greater in the first year this unit is coursed). However, these are very poor test results (between 0 and 3), which constitute at least 16% of students presented. However, this percentage is less than 23% compared to the course 99/00.

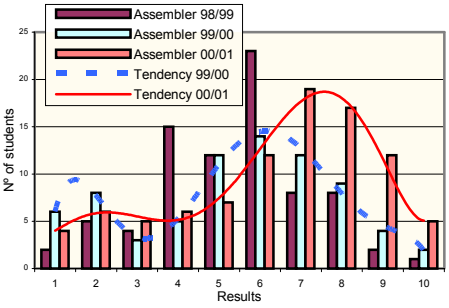

Figure 4. Histogram of the last three years assembler evaluation results.

5. Concluding Remarks

Our new methodology for the laboratory has shown that it is good to choose an RISC processor abstraction and a simulator for introductory computer architecture units. Although, it is also important that students follow their own pace and are actively responsible in this process. Sometimes it does not happen because students don't know how to do it. Our textbook [4] is an auxiliary tool to guide them in this way. Furthermore, in this new laboratory methodology, students advance more quickly, and see more contents in this unit, as input/output control and treatment of exceptions.

This methodology, as well as our textbook, has been adopted as a guide to other computer science programs at our University and at other Spanish universities. Evaluation results show that this is a good methodology improving instruction set architecture and assembler programming learning.

References

[1] ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 1991*. http://computer.or/education/cc1991.
[2] ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 2001*. http://computer.or/education/cc2001.
[3] A. Barenco, A. Ekert, A. Sanpera, C. Machiavello. *Un saut d'échelle pour les*

calculateurs. La Recherche, Nov 1996, http://www.qubit.org/intros/comp/comp.html.
[4] M.I. Castillo, J.M. Claver. *Prácticas guiadas para el Ensamblador del MIPS R2000 (in spanish)*. Dep. of Computer Science and Engineering. University Jaume I Editions. 2001, http://yan.act.uji.es/E38.
[5] A. Clements. *Selecting a Processor for Teaching Computer Architecture*. Microprocessors and Microsystems, may 1999.
[6] A. Clements. *The Undergraduate Curriculum in Computer Architecture*. IEEE Micro, pp. 13-22, may - june 2000.
[7] S. Dasgupta. *Computer Architecture - A Modern Synthesis*. John Wiley & Sons, 1989.
[8] E. Farquhar, P.J. Bunce. *The MIPS Programmer's Handbook*. Morgan Kaufmann, 1993.
[9] J.P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, 1998.
[10] J.L. Hennessy, D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3th edition, 2003.
[11] J.R. Larus. *MIPS. A R2000/3000 Simulator*. University of Wisconsin, http://www.cs.wisc.edu/~larus/spim.html.
[12] D.A. Patterson, J.L. Hennessy. *Computer Organization and Design. The Hardware/ Software Interface*. Morgan Kaufmann, 2nd edition, 1997.