International Journal of Engineering Research and Science & Technology

ISSN : 2319-5991
Vol. 5, No. 4
November 2016

www.ijerst.com

Email: editorijerst@gmail.com or editor@ijerst.com

International Journal of Engineering Research and Science & Technology

ISSN 2319-5991 www.ijerst.com
Vol. 5, No. 4, November 2016

Research Paper

LESSONS LEARNED IN TEACHING COMPUTER ARCHITECTURE

G M Prabhu[1]* and Simanta Mitra[1]

*Corresponding Author: G M Prabhu ✉ prabhu@iastate.edu

Teaching computer architecture to sophomores and juniors in computer science and software engineering is not easy. Many of them have the mindset that computer architecture is as irrelevant to their ability in finding a job as the study of Latin is to English majors. This mindset introduces extra challenges in teaching students the fundamental concepts. Our single most important outcome is to develop the student's ability to understand the fundamental principles so as to be able to make the appropriate tradeoffs in reaching design objectives. This paper describes lessons we have learned in teaching such a course: lessons both in computer architecture and in the pedagogy of teaching.

Keywords: Computer architecture, Tradeoffs, Rule of three, In-class activities

INTRODUCTION

Low-cost microprocessor-based systems were introduced in the early 1980s along with single-board computer systems. Students were able to get hands-on experiences with microprocessors by building simple systems, testing them, and interfacing them to peripheral devices. In the late 1980s and early 1990s more changes took place in computer architecture. Textbooks by John Hennessy and David Patterson (Hennessy and Patterson, 2011; and Patterson and Hennessy, 2013) were widely used and are still used in many computer science programs. The advent of the RISC-based microprocessor (Reduced Instruction Set Computer) took the focus of attention away from CISC-based computers (Complex Instruction Set Computer) such as the Motorola 68000 to high-performance, pipelined, 32-bit processors for the MIPS family. Their increase in performance makes it harder to give students the same hands-on experiences that students obtained in the 1970s when they could build a computer with readily available components.

The picture today is completely different. Now students have their own high-performance laptops and the role of the single-board computer in building systems is not as relevant. Assembly language (known as the poor person's C) which was used to write programs typically developed

[1] Department of Computer Science, Iowa State University, Ames, IA 50011, USA.

in high-level language, is now used primarily as a mechanism to illustrate instruction sets, addressing modes, and other aspects of the processor architecture. The rest of the paper is organized as follows. In Sections 2 through 4 we raise a question and provide a few answers. The questions are:

- Is it still relevant to teach computer architecture to computer science majors and software engineering majors?
- What should be taught in an undergraduate course?
- How do we teach this course?
- We offersome conclusions at the end.

RELEVANCE OF TEACHING COMPUTER ARCHITECTURE

A course on computer architecture is always in a state of flux because of developments in microprocessor design. Some see this topic as old-fashioned and want to replace it with material on web design or computer graphics.

However, if a program is accredited by the Computing Accreditation Commission of ABET, then it is required (as of now) to cover computer organization and architecture in the curriculum (ABET, 2014). But there are other important reasons that we believe computer architecture is as relevant today as it has been in the past.

The digital computer is central to computer science. The idea of a computer science or software engineering degree program that does not provide students with an insight into computers would be unusual in a university that is supposed to educate students, not merely train them for a job. Education is not the same as training and students should be taught more than being shown how to use commercially available computer software packages (Clements, 2014).

Computer architecture cannot be completely separated from software. Many processors are used in embedded applications and designing such processors and real-time systems requires one to understand fundamental architectural concepts and the limitations of processors. There are systems that are designed in the programming language C but are debugged using a logic analyzer that may display the relationship between interrupt requests and machine-level code.

It is useful to have a good background in computer architecture where students learn about the interaction between hardware and software. For example, to implement protection in a virtual memory system the hardware must have mechanisms and capabilities to transfer control from user mode to supervisor mode that permits execution of certain privileged instructions. Many real interfaces are still programmed at the machine level by accessing registers within them. Understanding computer architecture and assembly language can facilitate the design of high-performance interfaces (Clements, 2014). There are other reasons for teaching computer architecture. It incorporates important concepts that appear in other areas of the curriculum. For example, a finite state machine is used to describe the control function of a multi-cycle datapath for the MIPS processor. Students are also introduced to GPUs for game development and multiprocessing.

MATERIAL THAT SHOULD BE TAUGHT

In keeping up with the trends on how computers have evolved, computer architecture can focus

on instruction set architecture, pipelining, cache memories, Graphics Processing Units (GPUs), and parallel processors. Topics such as microprocessor systems design at the chip level and microprocessor interfacing should be phased out. However, the concept of tradeoffs is very critical and should be covered in the appropriate contexts.

Understanding how the different components of a computer system contribute to its overall performance is critical. Metrics like price/ performance ratio or cost/speedup ratio have to be used to make tradeoffs between different choices. For example, computer science and engineering students should be able to quantitatively determine whether it is cost-effective to double the size of the bus or increase the clock rate if these two options are presented to improve performance.

There is a wide variance in the material that is taught in an undergraduate course on computer organization and architecture as illustrated by the Table of Contents of the following textbooks.

The fifth edition of the classic textbook, "Computer Organization and Design" (Patterson and Hennessy, 2013) contains the following six chapters:

- Computer abstraction and technology
- Instructions: language of the computer
- Arithmetic for computers
- The processor
- Large and fast: exploiting memory hierarchy
- Parallel processors from client to cloud

The fourth edition of the textbook, "The Essentials of Computer Organization and Architecture" (Linda Null and Julia Lobur, 2015) contains the following 13 chapters:

- Introduction
- Data representation in computer systems
- Boolean algebra and digital logic
- MARIE: An introduction to a simple computer
- A closer look at instruction set architectures
- Memory
- Input/Output and storage systems
- System software
- Alternative architectures
- Topics in embedded systems
- Performance measurement and analysis
- Network organization and architecture
- Selected storage systems and interfaces

An instructor can, however, judiciously choose an appropriate topical outline based on the needs of computer science or software engineering students, any criteria from accrediting agencies (ABET, 2014), and topics which may be pre-requisites for other follow-on courses.

HOW WE TEACH THIS COURSE

The framework of our course is roughly divided into two parts: Fundamental principles (which do not change or change very slowly over time),and problem sets to illustrate these principles. The fundamental principles are distributed across the following areas: computer performance, processor datapath and control, memory hierarchy design, pipelining, and parallel processors and GPUs. All the principles are explained via problem solving. Students are trained to think by solving a lot of problems in class

and in homework. Problems are broken into smaller pieces and students are encouraged to work with the small pieces till they gain a full understanding of the underlying concepts.

The Rule of Three: A few years back we introduced a pedagogy which we call the rule of three. It works as follows. An example problem dealing with an important concept is first covered in lecture in some detail with all the steps clearly explained to students. The second exposure to the problem is through an in-class activity where students are asked to work in groups of two or three to solve the problem on their own. They are encouraged to ask for help from the instructor or the teaching assistants and they can verify their solution with the instructor's solution which is kept on a desk in the front of the class. After a day or two, the instructor e-mails the solution of the in-class activity to the students. The third exposure to the problem is via a homework exercise. Now when this concept is tested through a problem on the exam, student performance is much better than it used to be without the conscious application of the rule of three. Data to support this rule is given below through a direct assessment carried out for a course outcome via an exam problem.

Concept: Computing and comparing a cost/performance ratio to determine tradeoffs.

Course Outcome: Be able to apply mathematical principles (Amdahl's Law) and computer science theory to understand the tradeoffs between cost and performance in the context of computer architecture.

Exam Problem

You have a system that contains a special processor for doing floating-point operations. You have determined that 50% of your computations can use the floating-point processor. The floating-point processor is 40% faster than the regular processor.

- What is the overall speedup achieved by using the floating-point processor?
- In order to improve the overall speedup you are considering two options:

Option 1: Modifying the compiler so that 60% of the computations can use the floating-point processor. Cost of this option is $48,000.

Option 2: Modifying the floating-point processor so that it runs 100% faster than the regular processor. Assume in this case that 40% of the computations can use the floating-point processor. Cost of this option is $55,000.

Which option (1 or 2) would you recommend and why? Justify your answer quantitatively by comparing the [Cost/Overall Speedup] ratio for each option. Show all your work.

From Table 1 we observe that the percentage of students who scored 70% or more on the exam problem (i.e., at least 14 points) was 48.5% (34/

Table 1: Score Distributions with and Without the Rule of Three

Scores Obtained on the Exam Problem (Total is 20 Points)	Score Distribution Without Rule of Three (Total Taking Exam: 70)	Score Distribution with Rule of Three (Total Taking Exam: 105)
20 points	21	87
19 points	5	0
18 points	6	4
16 points	0	5
15 points	2	0
12 points	3	0
10 points	23	3
Below 10 points	10	6
Attainment of outcome	Unsatisfactory	Exemplary

70) when the rule of three was not used and 91.4% (96/105) when the rule of three was used. There are 5 levels in our rubric on attainment of outcomes: unsatisfactory, developing, satisfactory, good, and exemplary. A score of at least 70% on a problem implies an understanding of the underlying concept. If fewer than 50% achieve this threshold the attainment of the outcome is deemed unsatisfactory. The number of students taking the course was different in the two offerings, so percentages are used to assess the performance on the exam problem. Students were given a formula sheet containing Amdahl's Law so that did not become an impediment in understanding the concept of cost/performance tradeoffs.The attainment of the outcome from unsatisfactory to exemplary illustrates the effectiveness of the rule of three as a useful pedagogical tool in teaching concepts at the undergraduate level.

In-Class Activities: As described in the implementation of the rule of three, whenever an important concept is covered through the solution of a concrete example in lecture, an activity worksheet is distributed to the students to engage them into thinking about the concept on their own by solving a similar problem in groups of two or three. The lecture is usually 25 to 30 minutes long and the remaining time of 20 to 25 minutes is allotted for students to solve the problem (s) in the activity worksheet. An example of a difficult problem in an activity worksheet is reproduced below. This problem illustrates how a problem is broken into small pieces and how students enhance their understanding by systematically solving the smaller pieces.

Example Problem in Activity Worksheet

Suppose you have a load/store computer with the following instruction mix:

Table 2: Instruction Mix

Operation	Frequency	No. of Clock Cycles
ALU ops	40%	1
Loads	20%	3
Stores	15%	3
Branches	25%	4

- Compute the CPI for the above data. Show ALL your work.
- We observe that 30% of the ALU ops are paired with a load (i.e., they occur together), and we propose to replace these ALU ops and their loads with a new instruction. The new instruction takes 1 clock cycle. With the new instruction added, branches take 6 clock cycles. Compute the CPI for the new version. Show ALL your work.
- If the old clock is 20% faster than the new one, which version is faster and by what percent? Justify your answer quantitatively by showing ALL your work.

Student feedback on the in-class activities has been quite positive. They do much better in such a collaborative environment by working in groups and learning from their peers.

Emphasize Understanding, Not Memorization: Over the years that we have taught this course, we have been caught by surprise on many occasions to find out that more than a quarter of our students memorize solutions instead of understanding the underlying concepts. In the in-class activity example given above, the percentage of ALU Ops paired with a Load is 0.3 * 0.4 = 0.12 (30% of 40%). In the example worked out in class the numbers were different and the percentage of ALU Ops paired with a Load was 35% of 40% or 0.14. And yet many students just memorized the solution done in the class notes

and always used 0.14 to solve such a problem on the exam, regardless of the actual data that was given in the exam problem (which was different from that done in the lecture and the in-class activity and the homework).

Another instance in which memorization was rampant was in the control signals for the MIPS single cycle and multicycle datapath implementations. To discourage students from this sort of memorization, students were told ahead of time that the datapath diagrams that would be given to them in the exam would be different from those in the handouts given in class. They were told explicitly not to memorize the values of the control signals but to ask us and our teaching assistants for help in understanding the concept of control. We even went to the extent of stating that anyone who wrote down a memorized solution would receive zero partial credit on the exam. And yet, a significant number of students wrote down the memorized solutions from the class notes for the exam problems on datapath and control.

Using Technology in the Classroom: There seems to be a preoccupation with using technology in the classroom. Certainly there is much that can be done with technology to improve learning. But we should stop blindly promoting the use of high-tech in education. Poor pedagogy does not become good pedagogy merely by draping it in technology.

One good outcome of technology is the use of simulators to enhance learning based on visualizations of different architectures. There are no adequate tools to visualize the dynamic interaction of software and hardware during program execution at the machine level. Existing tools are mainly dedicated to debugging by professional programmers. A number of educators, motivated by the intuition that visualization can contribute to overcoming their teaching difficulties, have resorted to the use of simulators to meet their teaching needs (Yehezkel *et al.*, 2007).

In this semester's offering of the course, students are not allowed to use a laptop or tablet in our class and we have asked them to silence their cell phones. Instead of using technology as a crutch we have asked students to take their own class notes in a notebook so they form a basis for understanding what they are learning. These class notes will be evaluated three times during the semester and weighted at 10% of a student's overall score. We have yet to determine the efficacy of doing this but early indications are that it is helping students to stay alert in the 8 AM time slot during which this course has been scheduled in Fall 2016. If there is any sign that this is helping students, it is the fact that the Exam 1 average was close to 80 points out of 100.

CONCLUSION

We conclude by providing a brief summary of answers to the three questions we raised in the Introduction.

We take the view that a course in computer architecture can provide a suitable forum for incorporating some fundamental principles (such as tradeoffs) in the computer science and software engineering curricula. Computer architecture is relevant because it provides bottom-up support for the top-down methodology taught in high-level languages.

With demands from the multimedia world, Intel's MMX and SSE instruction sets, and smart disk drives, a case can be made that we should

teach computer systems architecture rather than a processor-centric course on computer architecture. This is an idea that can be explored in choosing topics to include in this knowledge area. As the market becomes more demanding with developments such as NVIDIA's Compute Unified Device Architecture (CUDA), it is extremely useful to obtain input from organizations such as the ACM and IEEE which articulate from time to time the various knowledge units that should go into courses in a computer science curriculum (ACM, IEEE, 2013).

We have found our pedagogical approach such as the rule of three and in-class activities to be quite successful in imparting fundamental concepts. However, we are still struggling with ways to emphasize understanding and not memorization.We have succeeded with about 75% of the students but not with all of them. It is our hope that by not rewarding memorization we will encourage all our students to learn to think and perhaps focus on understanding the fundamental concepts.

REFERENCES

1. ABET (2014), http://www.abet.org/wp-content/uploads/2015/05/C001-15-16-CAC-Criteria-03-10-15.pdf
2. ACM and IEEE Joint Task Force (2013), "Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science".
3. Clements Alan (2014), *Computer Organization and Architecture*, 2nd Edition, Cengage Learning.
4. Hennessy John and David Patterson (2011), *Computer Architecture: A Quantitative Approach*, 5th Edition, Morgan Kaufmann.
5. Null Linda and Julia Lobur (2015), *The Essentials of Computer Organization and Architecture*, 4th Edition, Jones and Bartlett.
6. Patterson David and John Hennessy (2013), *Computer Organization and Design: The Hardware/Software Interface*, 5th Edition, Morgan Kaufmann.
7. Yehezkel C, Ben-Ari M and Dreyfus T (2007), "The Contribution of Visualization to Learning Computer Architecture", *Computer Science Education*, Vol. 2, No. 17, pp. 17-27.

International Journal of Engineering Research and Science & Technology
Hyderabad, INDIA. Ph: +91-09441351700, 09059645577
E-mail: editorijlerst@gmail.com or editor@ijerst.com
Website: www.ijerst.com