"What does a CPU have in common with a fast food restaurant?" A Reflection on Emphasizing the Big Ideas of Computer Science in a Computer Organization Class

Craig Zilles
Department of Computer Science, University of Illinois at Urbana Champaign
Urbana, IL 61801 zilles@cs.uiuc.edu

***Abstract* - While each class in a computer science curriculum serves to teach the principles and core knowledge of one domain, it is equally important that our students receive a deep understanding of the central ideas of the field that cut across sub-disciplines. In this respect, a class on computer organization provides an ideal context for concretely demonstrating some of the most important ideas in computer science. This paper describes how, with little effort, discussions of abstraction, indirection, and Turing completeness can be introduced into lectures on computer organization. We also discuss explaining the broader applications of two architecture-centric ideas, caching and pipelining. We present data from pre- and post-tests on our students learning of these concepts, demonstrating the relative difficulty of these ideas for students and identifying some of the sources of student misconceptions.**

Index Terms – Computer Science, Computer Organization, Concepts, Education.

INTRODUCTION

Like every field, there is a collection of ideas that are central to Computer Science. It is a deep understanding of these ideas and the ability to apply them appropriately in the construction and analysis of computing systems that is the signature of thinking like a computer scientist. These ideas are important enough that they should be touched on multiple times throughout the curriculum to ensure the students achieve a deep understanding.

A number of these ideas are well suited to be introduced and/or emphasized in the context of a class on computer organization. In particular, this paper focuses on five ideas, three of which (abstraction, indirection, and Turing completeness) are core CS ideas where our goal is to provide a concrete example of these somewhat abstract ideas, and the other two (caching and pipelining) are computer architecture staples where our goal is to demonstrate how these ideas apply to computing beyond architecture. We discuss how we present these ideas in a course at the University of Illinois (CS 232: Computer Architecture II), which covers assembly language programming, machine organization, and memory and I/O systems. This course is generally the 4th or 5th course our students take in our curriculum.

As part of this paper, we describe an initial assessment of student learning of these concepts. This assessment was motivated by research in Physics education literature [1], which has since been repeated in other fields (*e.g.*, [2]), that demonstrated that students with only a vague conceptual understanding can still perform well on traditional problem-solving exams by pattern matching to previously studied problems. By asking students conceptual-oriented questions (*e.g.*, [3])—in addition to problem-solving oriented questions—we can more readily assess the learning of these concepts, identify common misconceptions, and change our teaching to address these misconceptions.

After a discussion of the assessment methodology used (in the next section), we discuss each of the five ideas in turn. For each idea, we discuss how we integrate the concept into our course, and our experiences with teaching the idea, including any important student pre-conceptions and common misconceptions.

METHODOLOGY

The results we present in this paper are based on assessments of understanding of the concepts taken both before and after instruction. We collected data for 4 of the 5 concepts (all except Turing completeness). All assessments were scored and interpreted by the author.

Our pre-test data comes from a "background knowledge" assessment given to the students on the first day of class during the Spring 2005 semester. For each concept, the students were asked to define it and provide one example, if possible. 95 students returned the survey. A number of students left some concepts blank. There are two possible reasons for this: 1) they had no idea (or were unconfident), and there was no motivation to guess (as there was no credit to be earned), and 2) they lost interest or ran out of time. Two pieces of evidence supports the first conclusion: 1) frequently when a respondent wrote about N concepts, it wasn't the first N, and 2) each concept had a different number of "no responses" and the number was correlated to the number of incorrect responses on the post-test.

The post-test data was collected as part of the final exam during the Fall 2004 semester. For each of the four concepts, the students were asked to provide a definition and an example **not** related to computer architecture. 103 students took the exam and there was sufficient time allotted that there was almost a 100% response rate on all four concepts.

As the pre- and post-test data were collected in different semesters, we cannot conclusively relate the two data sets to each other. Nevertheless, the populations are large enough and of sufficiently similar make-up that such comparisons are not unreasonable.

CONCEPT 1: ABSTRACTION LAYERS (ENCAPSULATION)

The notion of an abstraction layer is a fundamental idea in computer science, in that it enables the decomposition of large complicated systems into a number of smaller components that interact through well-defined interfaces. A terse definition of this concept is "separating interface from implementation." We discuss this idea in CS 232 in the context of instruction set architectures (ISAs), which serve as an abstraction layer between the software and the hardware. We emphasize that this idea is fundamental to the business model of both hardware and software vendors, enabling each to sell their wares independently. In particular, hardware vendors can sell new (faster) hardware without a customer having to modify or recompile their software.

I've found that this is not a difficult concept for the students to learn, perhaps in part due to some previous exposure to the concept. Given the usage of the word "abstract" in non-technical English, most students have an association with the word "abstraction". In the pre-test, one-third of the provided definitions for the word "abstraction" were suggestive of the concept of abstraction layers, often including phrases like "hiding implementation (low-level) details". For the remaining students the word holds connotations of simplification (*i.e.*, omitting rather than hiding details, often for analytical or pedagogical purposes) or of symbolic representations and "syntactic sugar" (*e.g.*, block diagrams and "describing in terms of another language what goes on in a base language"). Other definitions touched on "making more generic" (in the LISP macro sense), decomposition/composition (which abstraction layers enable), references to hardware abstraction layers, and extraction of data (extraction is synonymous with Webster's first definition of abstraction).

Because I feel strongly that the students learn this idea, I tell them explicitly that I will ask the following question on the first mid-term:

"What is an abstraction layer? How does it relate to ISAs?"

This serves to give the students an example of the kinds of conceptual questions I ask on exams and reduces exam anxiety. It also results in over 90% of the students being able to answer the question in a satisfactory manner. More importantly, the students retain the knowledge. At the final exam, almost three months later, over 80% of the students can both define abstraction layers and give one example not related to computer architecture, which demonstrates that they are not just parroting back what I have explicitly taught them. The most common example the students use is that of a car (*e.g.*, "can drive it; don't know how it works"). Another nice student example is "not micro-managing: telling what you want, not how to do it."

CONCEPT 2: INDIRECTION

In contrast to abstraction layers, our students have a much more difficult time achieving a deep understanding of indirection. While indirection is unquestionably an important concept in computer science (*i.e.*, "all problems in Computer Science can be solved by another level of indirection" *–David Wheeler*), there seems not to be a canonical definition of it. Definitions that capture the power of the idea (*e.g.*, "Indirection is the ability to reference something using a name, reference, or container instead the value itself") are abstract and difficult to grasp, while more concrete definitions (*e.g.*, "Manipulating data via its address") do not do justice to the broad implications of the idea. In retrospect, it is not surprising that students have a difficult time with this concept.

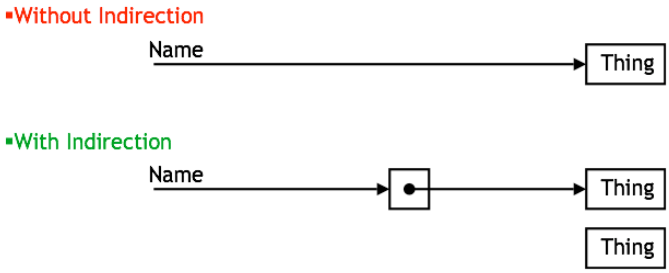

FIGURE 1
AN ILLUSTRATION OF HOW INDIRECTION ENABLES RE-BINDING A NAME TO A THING THROUGH AN INTERMEDIATE VARIABLE.

We introduce the concept of indirection in CS 232 in a discussion of virtual memory. We first discuss the concept in the abstract, as providing a flexible mapping between a name and a thing (as shown in Figure 1), and then provide students with some examples they may already be familiar with, including: US postal service mail forwarding, domain name service (DNS), and how their cell phone number gets mapped to the closest base station. Virtual memory illustrates indirection in two ways: 1) a level of indirection provides a flexible mapping between the virtual addresses that programs use and the physical addresses in a machine, and 2) multi-level page tables provide a second example of indirection, enabling only a subset of the page table to be allocated and resident in physical memory. The multi-level page table also provides a good example of the limitations of indirection ("all problems in Computer Science can be solved by another level of indirection... except too many levels of indirection") where caching, in the form of the translation look-aside buffer (TLB), eliminates much of the performance cost of indirection's functionality.

The pre-test demonstrates that indirection is an idea that the students have little familiarity. Only 12% of the students had definitions suggestive of the CS meaning of the word (*e.g.*, "referencing data by having a variable point to it") and over 75% of the students had no response.

The post-test results were better, but noticeably worse than the other three concepts tested. Only 44% of the students were able to both define indirection and provide a non-computer architecture example of it. Another 30% were able to define it, but not provide an example, suggesting to me that they had a fragile understanding of the concept. Of the students that could not define indirection, one of them could draw the picture in Figure 1, but stated that he did not know what it meant. We are currently exploring how to better communicate this concept.

CONCEPT 3: TURING COMPLETENESS

The concept of Turing completeness is a fundamental principle of computability theory. A machine is Turing complete if it has computational power equivalent to the theoretical universal Turing machine. As modern computers are generally Turing complete, the practical implication of this idea is that any computer can be emulated on any other computer.

Interestingly, this is an idea that is very unintuitive for our students (who generally have not taken an automata class before CS 232), as it conflicts with their first-hand observations. Our students are well aware of the notion of binary compatibility: that an x86 program won't run on a Mac computer. As I discuss below, many students interpret this incompatibility as a fundamental one and not a merely a consequence of the required software engineering to build an emulator.

We discuss Turing completeness in two circumstances in CS 232. The first occurs in a discussion of exceptions; we use software emulation of unsupported instructions (*e.g.*, running code with SSE instructions on a pre-SSE PentiumPro) to walk through how non-terminating exceptions can be handled. In this discussion, I ask, "Can we always use this technique to emulate instructions added by ISA extensions?" Understanding Turing completeness makes it clear that any (deterministic) instruction can be emulated, but (unlike most questions I ask in class) this question goes unanswered. In answering this question, I explain (conversationally) the notion of Turing completeness as it relates to emulation.

The topic resurfaces a week later, when we start talking about performance. As an introduction to performance, I do a in-class brainstorm, where I have the students list a number of ways computing systems differ, where I provide them a diverse set of example computer systems to consider (Figure 2); this is a very effective in-class activity even in large (100+ student) lecture contexts. I preface this with the question, "Do computers differ functionally? Can some computers compute things others can't?" In light of our discussion on Turing completeness, some students are now comfortable with this idea and can explain why this isn't the case (assuming memory is not a limitation). But, when we begin to identify the ways that computers differ, a frequent contribution is "compatibility" in various guises. This is surprisingly ingrained in the students, to the point where one student used as an example "for example a PlayStation2 can't run Linux", which in fact has been done.

FIGURE 2
EXAMPLE COMPUTERS TO FACILITATE AN IN-CLASS BRAINSTORM OF THE WAYS COMPUTERS DIFFER FROM EACH OTHER.

While my experiences on this concept are only anecdotal, it appears that, for many students, the notion of Turing completeness conflicts with closely held beliefs and knowing this facilitates the teaching of this concept.

CONCEPT 4: CACHING

Caching, defined as "a process in which frequently accessed data is kept on hand, rather than constantly being accessed from the place where it is stored," is one of the most important ideas in computer architecture, and, thus, is a focus of any course on computer organization. Our additional goal was that students internalize the concept so they can recognize and conceive its use in non-architecture settings.

From the pre-test, we find that most of our students have seen the word cache before (over 80% respond), and about one-third of them can define it satisfactorily. Those that cannot define it understand some of its connotations: it relates to data storage or memory, it is a temporary storage mechanism, and it makes things faster. Many know that it is something that is part of modern processors; others encounter the term in the context of a web browsers and/or mail clients as many of the student responses suggest storage of data for later offline use.

In CS 232, after introducing the basics of caching, I have the students identify instances of caching in their everyday lives. One that is very memorable to the students (many of them used this example on the post-test) was how a refrigerator reduces the frequency at which one needs to go to the grocery store. Overall, we find caching to be the easiest concept for the students to learn, with over 90% of the students able to both define and give a non-architecture example of it.

CONCEPT 5: PIPELINING

Pipelining is the process of breaking a task into a sequence of sub-tasks and executing these sub-tasks each on their own resource. Pipelining is a particularly effective technique for implementing processors, because it decouples instruction throughput from latency of instruction execution. Again, in

teaching this concept, I hope to emphasize that it is not a hardware-specific idea by presenting a broad range of examples. Examples we touch on in class include the canonical laundry example, pipelining the packets of a large TCP/IP file transfer, the 3D graphics rendering pipeline (a software pipeline), and drive-through lanes at fast food restaurants.

Pipelining is also relatively easy for the students to grasp. While about a quarter of the students coming into CS 232 are able to define/describe pipelining (a few students report it being mentioned in the pre-requisite logic design class), 86% of the students could both define and provide a non-computer example of pipelining. I think this is in large part due to the memorable examples. In fact, most of the students that could not precisely define pipelining could still provide an example of it.

The major confusion that the students had with pipelining was distinguishing it from multiprocessing. This confusion is easily addressed by specifically contrasting the two ideas. One example that I use to distinguish the two is checkout lanes at grocery stores. Opening a second checkout lane is multiprocessing; adding a bagger to a checkout lane is pipelining. From this example it is clear that the ideas are orthogonal and complimentary.

This confusion with multiprocessing is one that some of the students have entering the class. For at least a few students, this confusion comes (at least in part) from reading the specifications of graphics processors, which often tout the number of pixel processing pipelines, leading them to associate the word with multiprocessing. Other students associate pipelining with streamlining (*i.e.*, the act of optimizing) and moving data (*e.g.*, Unix pipes).

CONCLUSION

To think like computer scientists our students need a firm understanding of the central ideas of computer science. In this paper, I've discussed how some of these ideas (abstraction layers, indirection, Turing completeness) can be introduced and/or emphasized in the context of a course on computer organization and how to emphasize the broader applicability of the two computer architecture ideas (caching and pipelining). Of these concepts, the students have the most difficulty with indirection and Turing completeness. While abstraction layers, caching, and pipelining are concepts that have memorable real world examples, indirection and Turing completeness are more abstract concepts. In particular, many students find Turing completeness incongruous with their personal experience.

REFERENCES

[1] Thalloun, I. A., and D. Hestenes, "The Initial Knowledge State of College Physics Students", *Am. J. Phys,* 53 (11), Nov 1985.

[2] Jacobi, A., J. Martin, J. Mitchell, and T. Newell, "A Concept Inventory for Heat Transfer", *Frontiers in Education,* Nov 2003.

[3] Hestenes, D., M. Wells, and G. Swackhamer, "Force Concept Inventory", *The Physics Teacher,* 30, Mar 1992.