

# Teaching Computer Architecture in Introductory Computing: *Why? and How?*

Kris D. Powers

Dept. of Mathematics and Computer Science  
Berry College  
Mount Berry, GA 30149 USA  
kpowers@berry.edu

## Abstract

This paper discusses our rationale for teaching the fundamentals of computer architecture early in the computer science (CS) degree program. We also describe an active learning technique that we use to help make this material accessible to our introductory CS students.

*Keywords:* computer architecture, computer science education, active learning

## 1 Introduction

*Computing Curricula 2001* (ACM/IEEE Task Force, 2001; hereafter *CC2001*) identifies 14 *areas* in the discipline of CS, and specific *units* within these areas that comprise the core body of knowledge for a CS degree program. These core units represent the coverage essential to the education of any student in a CS degree program; they are not meant to be viewed as constituting a complete curriculum. *CC2001* also provided a metric in terms of lecture *hours* for comparing the minimum amount of time required for covering each core unit.

Of the 14 identified areas, the distribution of lecture hours would seem a reasonable indicator of the centrality and relative importance of the area to the CS discipline, as recognized by the *CC2001* Task Force. It is interesting to note that only 5 of the 14 areas carry in excess of 30 hours (with the next closest at only ~ 20 hours). Further, only discrete structures (DS) at 43 hours and programming fundamentals (PF) at 38 hours were allocated more time than the architecture and organization (AR) area at 36 hours. Thus, there seems to be broad consensus that understanding the low level organization and operation of a computer remains a relatively important aspect of a CS degree.

We agree. Indeed, it has been our experience that a solid understanding of the lower levels of the virtual machine hierarchy provides essential underpinnings for the effective use of the upper levels. An important issue to the development of this understanding is the point in the CS degree program at which it is initially engaged. In this work we present our rationale for introducing this topic as

early as possible. And, as a practical response to those that would question the efficacy of such an approach, we discuss an active learning laboratory for our students' first exposure that we have found to work especially well.

## 2 Background

### 2.1 Curricular recommendations

While the current recommendation for lecture hours in the AR area remain high when compared to other areas, it should be noted that the *CC2001* curriculum guidelines generally reflect a reduction in the time devoted to AR topics. Indeed, *Computing Curricula 1991* (ACM/IEEE Task Force, 1991; hereafter *CC1991*) includes a whopping 59 hours in the AR area. Further, 10 out of 12 of the sample curricula presented in *CC1991* include two courses in this area, typically a course focused on assembly language programming followed by a course in computer architecture. In the decade that followed the publication of *CC1991*, however, the prominence of the AR area within the discipline diminished, making way for new areas (e.g., the WWW) and areas of growing importance (e.g., networks). Many CS degree programs have folded coverage of assembly language programming into a single architecture course. This reality is reflected in the approaches to the intermediate courses in *CC2001*. These approaches consistently use a single course to cover the necessary knowledge units in the AR area. Thus, while the other 2 areas with a high number of knowledge units (PF and DS) are recommended to be taught over two or more courses, AR is typically expected to be taught in one. Further, courses in AR can often be deferred until the upper division (junior/senior) level.

### 2.2 Educational theory

#### 2.2.1 Spacing and Spiralling

The terms given to the educational concepts of spiralling and spacing are remarkably lucid. The well-known concept of *spiralling* refers to revisiting previously discussed topics and expanding upon them at a higher intellectual level. The concept of *spacing* seems less widely known. The idea is simply that of spreading learning out over a longer period of time. Note that the issue is the total time frame allotted for achieving the learning objective - not the time spent on task. The spacing effect is often linked to spiralling when the development of a learning goal is spaced at multiple levels. However, the spacing effect is novel among

educational theories: we actually have experimental evidence that it increases learning! The spacing effect is “one of the most dependable and replicable phenomena in experimental psychology... remarkably robust ...(and) truly ubiquitous in scope.” (Dempster, 1988, p. 627) Based on such studies, the “spacing effect would seem to have considerable potential for improving classroom learning, yet there is no evidence of widespread application.” (Dempster, 1988, p. 627)

### 2.2.2 Constructivism and Preconceptions

The educational theory of *constructivism* holds that while there is a physical reality, we can never say that what we know is the truth because all of our knowledge has been constructed from our own personal experiences and social interactions in a particular cultural setting. Since no one's experience is complete, no one's knowledge is complete. Further, any new knowledge our students construct in response to new experience will be incorporated into the framework of knowledge they have already constructed. The old knowledge affects the new.

The theory of constructivism has a number of important implications for methods of teaching. The knowledge the students have already constructed is based upon previous experiences. If the learning we attempt to provide has no basis in experience, it has little chance of modifying that which the students already “know.” So one implication is that our teaching must be *experiential* to be effective. Also, because their perceptions explain what they have experienced, students believe that this knowledge is correct. Even if the ideas seem ludicrous, they are merely naive – based on incomplete experience and a lack of social interactions that would challenge them. We as educators must explicitly confront these *preconceived notions* before they can be dispelled. Finally, knowledge is constructed in a *social setting* influenced by the instructor. Within this setting students must be provided with an opportunity to form new knowledge in cooperation and interaction with their peers.

Of these implications for methods of teaching, the difficulty of preconceived notions is probably the least actively addressed in the teaching CS. This issue is readily demonstrated by the common student perception that CS *is* programming. (Other examples of the inaccurate notions that complicate CS teaching can be found in Powers and Powers (2000)). However, the most important preconceived notion that a student brings to the introductory CS classroom is their fundamental computer concept. A critical factor here is the extent to which student concepts may be considered “effective” or “viable” for explaining the totality of their (limited) experiences with computing machines. Ben-Ari (1998) considered generally the theory of constructivism in the context of computer science education. In this work he contended that “...the lack of an effective, if flawed, model of a computer is a serious obstacle to learning CSIS.” (Ben-Ari, 1998, p. 259) And: “If the student does not bring a preconceived model to class, then we must ensure that a viable hierarchy of models is constructed and refined as learning progresses.” (Ben-Ari, 1998, p. 260)

### 3 Why teach AR in the introductory sequence?

Our rationale for suggesting early coverage of AR topics follows, to a great degree, directly from the educational theories.

If students do indeed enter their first CS classroom without an effective model of a computer and no effort is made to rectify the matter, then they will continue to construct further knowledge based upon a weak and potentially flawed intellectual scaffolding. The student will be left susceptible to later intellectual challenges that can topple their mental framework. Thus, it is essential that students be aided in the development of a viable model, and that any pre-existing and possibly erroneous computer concepts be fully challenged as early as possible in a CS degree program.

For instance, in order to discuss programming language semantics, many educators rely upon an effective model of memory as a sequence of numbered storage slots. The contents of a specific location can be accessed by providing the associated number, and those contents can be replaced or copied as needed. A student may adopt a significantly different model in which values are *added to* (as opposed to *replaced*) in a location or values are *moved* (as opposed to *copied*). The model actually employed by the student will be tested when more sophisticated concepts such as array access or pointers/references are later encountered. If the initial understanding is flawed, the challenge of incorporating the later topics may require re-learning the initial concept at a fundamental level.

In addition, the opportunity to fully engage material on a more sophisticated level may also be lost. If students develop an early understanding of lower level operations, then the study of AR as well as other areas can be enhanced and deepened. For example, discussions of activation records and the run-time stack in a programming languages course can be engaged at a higher intellectual plane when a basic background in AR is presumed.

Another part of our rationale stems from the intrinsic difficulty in understanding the lower level operation of a computer. It seems highly unlikely that a typical undergraduate student can suitably grasp it in “one take.” Indeed, a primary implication of spiralling is that our students need the chance to engage topics at a variety of intellectual levels. As students re-encounter a topic at later points, they can engage it at a higher level of intellectual sophistication. Not only is previous learning reinforced, but also new learning is supported by the student's pre-existing conceptual framework. Thus, there would seem to be the potential for improved learning by spreading the learning of a concept over multiple courses at multiple levels of the CS curriculum.

The knowledge areas that are most central to the CS discipline, i.e., those warranting the most core hours, would especially seem to warrant coverage at multiple points in the CS curriculum. The topics taught in the PF and DS knowledge areas are naturally revisited and reinforced throughout a CS degree program, as

computational problem solving is applied in a variety of contexts. Meanwhile, for AR topics, this is often not the case. More often than not, there is single course in which AR topics are addressed. And, often this course happens significantly later, if not at the end, of a CS student's degree program. Such an approach fails utterly to exploit the potential of spacing and spiralling.

## **4 How to teach AR in the introductory sequence?**

### **4.1 Finding a place for AR topics in the introductory sequence**

It should be noted that a significant number of CS programs already do cover AR topics early in their curricula. One common approach is the inclusion of an introductory CS0 course that presents a breadth-first overview of the CS discipline. Less common is the adoption of a breadth-first approach generally, for the entire curriculum. Programs without a breadth-first component incorporate AR topics early by including the topic in the traditional CS1 course. This is especially easy to accomplish if the suggestion for a 3 course introductory sequence made by the *CC2001* Task Force has been adopted. Part of the expanded time frame can be appropriated for basic AR topics. Further, many programming texts do begin with a section or two that address the fundamental operation of a machine. These parts are often glossed over or skipped entirely. We suggest that they be included (and potentially expanded upon) as an essential part of the CS1 course, and encompass topics including the fundamentals of machine/assembly language and the fetch-execute cycle.

### **4.2 Approaches for teaching AR concepts at the introductory level**

Certainly there are many well-known approaches and tools for teaching these concepts at the introductory level of a curriculum. For example, the CS0 text of Dale and Lewis (2002) includes entire chapters on computing components and low-level programming, and is supported by a CPU simulation in the associated lab manual by Meyers (2003). We have also used tools like Skrien's *CPUSim* which allows the user to define the organization and microoperations of the CPU to be simulated. Indeed, simulation tools for visualization of CPU function have been in use for many years (see, e.g., Decker and Hirshfield (1990)). However, while such tools do vastly improve on blackboard demonstrations of CPU function, we have found that the introduction of an active learning component significantly aids student understanding.

#### **4.2.1 The Living CPU**

*Active learning* generally refers to pedagogies that embrace the notion that students actively construct their own knowledge; the task of the teacher is to provide the opportunity for this to occur. (See, e.g., Silberman (1996) or Johnson, Johnson, and Smith (1998) for expositions on the topic.) While CPU simulators have great potential to improve student understanding, they often fall subject to passive use. Students may be encouraged to work actively and/or cooperatively; e.g., "predict the effect of the next step with lab partner – list everything that you expect to see change"), but such approaches may falter in the absence of direct instructor intervention and support. Students' reactions of: "I don't know why that is happening next!" can be common. In our experience, such reactions are indicative of and exacerbated by the complex appearance (to the novice's eye) of the environment. There is also apparent a singular lack of ownership in the functioning of the simulator, enhancing student disconnect from the tool.

The "Living CPU" lab exercise is designed to help the students deduce the functioning of the CPU through Socratic-like questions and answers, and then bring it to life with student actors. Such an approach leads to direct student ownership of the concepts: students not only play the parts, but they have "written the lines" as well. And, as is the hallmark of any active learning exercise, the students are actively constructing their own knowledge.

As prerequisite to this activity, we have usually already covered binary numbers and data representations. As a hook into the activity, we often begin by looking at a computer advertisement, and have students describe the basic parts of a computer: CPU, RAM, disk drive, etcetera. Our only goal at this initial stage is to extract a simple view of a computer as at least 2 parts: processor and memory.

From there, we proceed to lead students into filling in certain details, including the need for and fundamental steps of the fetch-execute cycle as well as more detailed parts of the machine, including: registers, ALU, control and bus. Our approach is guided by the following ground rules:

- We present a series of questions.
- We never answer our own question, although introducing an additional question is permitted.
- No one student can answer consecutive questions.
- As many students as possible are included.

Each question has a response objective that is needed to progress. Table 1 gives the initial sequence such an inquiry might follow. Note that as students discover the need to reference certain parts or actions, standard terminology is introduced.

Inquiry	Response objective
What happens when you double click on your term paper document on the desktop?	Word processing program and term paper data are loaded into memory.
What is a program?	Instructions to the computer
How are instructions carried out?	By the processor
How do instructions get from memory to the processor?	Wires (introduce the term <i>bus</i> )
What happens to an instruction when it gets to the processor?	It gets performed (introduce the term <i>executed</i> )
What kinds of processing (i.e., instructions) can the machine do?	Arithmetic (introduce the term <i>ALU</i> for that part of CPU)
Here are two really big numbers, add them... how did you do it?	Wrote them down (scratch pad in CPU; i.e., introduce term <i>data registers</i> )
How does the machine decide whether to, e.g., add versus subtract?	Recognizes different kinds of instructions (introduce the term <i>control</i> )

**Table 1: Initial inquiry sequence**

While the listed response objectives may seem ambitious, we have yet to encounter a class that cannot resolve them. Indeed, it would seem from our experience that the conceptualisations of a computer as a processor of instructions, and memory as a residence for both instructions and data are surprisingly wide spread. On the flip side, however, we rarely encounter student understanding that transcends these superficial descriptions. And, it should be noted that these conceptualisations, while wide spread, are not yet entirely ubiquitous.

After suitable inquiry, the operation of the machine as described by the students is summarized, and new terminology can be attached to the identified parts. Table 2 gives an example of the script that might have been generated up to this point. Invariably there are holes in the final product that the instructor must help fill. One concept that is typically weak, if recognized at all, is the idea of memory as numbered storage slots. The need to somehow specify which locations in memory are involved in a data transfer is usually clear; but not necessarily how to do it. The specification of a storage slot by using an associated number seems to require explanation. And, of course, the idea of representing the operation and operands in a machine code format like:

`<operation> <result> <op1> <op2>`

is totally non-intuitive and must also be presented by the instructor. We assume a simple memory-to-memory

architecture, and so `<result>`, `<op1>`, and `<op2>` are all memory locations. We use 4 op-codes for addition, subtraction, multiplication, and division (assigned the codes 0-3, respectively). The operation performed is:

`<result> ← <op1> <operation> <op2>`

Certain other holes in the script, like the need for a program counter or instruction register are nearly always left for the students to discover later in the exercise.

<p>Transfer an instruction from memory to the processor using the bus.</p> <p>Have the control figure out what kind of instruction it is.</p> <p>Transfer the needed data from memory to the scratchpad in the processor using the bus.</p> <p>Perform the instruction (add, subtract, etc.) using the ALU</p> <p>Transfer the result from the scratchpad in the processor back to memory using the bus.</p>
--

**Table 2: A student generated “script”**

Once students have developed their own basic algorithm for the execution of machine language instructions, they are asked to bring it to life. We have been most successful when we “set the scene” by incorporating a collection of simple props/manipulatives. These include:

- large envelopes for storage cells; registers consist of a single envelope labelled with the register name while memory consists of an rectangular array of numbered envelopes
- index cards for storage cell contents; note that no storage cell is ever “empty” – every envelope always contains a card
- markers for changing the contents of storage cells
- a metronome for the computer clock
- labels for each actor identifying their role

The script for the CPU processing is carried out by actors assigned to the following roles:

- control – calls out the required actions; initially at least, a starring role for the instructor
- ALU – performs the needed computation using the scratch pad registers; i.e., it can look at the 2 operand registers (read the index cards in the envelopes for the 2 operand registers) and put new value in the result register (i.e., write a new value on the index card in the envelope for the result register)
- bus – transfers (i.e., carries) copies of storage cell contents (i.e., index cards) from one place (i.e., envelope) to another
- memory (controller) – copies values to/from memory storage cells (i.e., updates the index card in the envelope for a storage location or carried by the bus)

It is important that the acceptable actions for each player be carefully and exactly delimited. In particular, it is important to note that no actor except the bus is permitted to “hold” data for any length of time – it must be “stored” (put on an index card in an envelope). It is also important to position memory a suitable distance across the room from the CPU, so that the work of the bus can be clearly recognized, and so that the actors do not end up bumping into each other.

When the students are initially asked to “fetch-execute” an instruction, the need for a program counter is immediately discovered (“Er... uh.. which one?”). The need for the instruction register is recognized just as quickly when the bus returns from memory with the first instruction, and needs a place to put it. The execution phase then usually proceeds in a straightforward fashion, as the control:

- determines the operation for the instruction
- tells bus to transfer data from memory to the operand registers
- tells ALU what operation to do
- tells bus to store the contents of the result register back to memory

Finally, when the demonstration is attempted for a second instruction, the need for updating the program counter is deduced and incorporated into the script.

The demonstration is run several times, giving different sets of student the opportunity to both play a role, as well as to observe and to make notes. As the comfort level in the classroom increases, typically the speed at which the actors accomplish the steps increases as well. Noting this provides a chance to bring up the idea of the clock and its role in governing the speed at which the CPU operates. While attempting to run the demonstration with each step performed at set rate (as dictated by either a metronome or a student taking on the role of clock) is usually not very successful, it is often illustrative. For example, students often recognize the need for the longest step of the process to dictate the speed of the clock. (Setting the clock speed to an extra fast speed to simulate “over clocking,” while usually not illustrative, is often fairly amusing!)

## 5 Conclusions

It is interesting to note that our “Living CPU” lab exercise begins by questioning students about their initial computer concepts. The extent of their understanding typically ranges from shallow to non-existent; however, the ideas that are expressed are usually fairly accurate. Thus, while our students are lacking in their computer concept, we do not seem to be combating erroneous impressions in our introductory CS classroom. We believe that by strengthening these concepts early, later misconceptions in AR as well as in other areas of CS are avoided.

The “Living CPU” exercise described is suitable for introductory discussion of the fetch-execute algorithm and low level machine organization. Typically, we follow

it in the same course with further experiences using a CPU simulator. The architectural model thus developed is exploited throughout the course as it is used to explain fundamental programming concepts. For example, discussions of variables as entities associated with storage locations and, in fact, of any issues related to addressing modes seem much more effective when a low-level model can be referenced. (This has been especially useful in discussing object variables and the storage of references early in a Java based course.)

Our students also re-encounter the AR area in their later studies through a required 4 hour, lab-based course. Our informal observations at this level support the idea that our students’ eventual understanding of the low level machine operation is deepened by our intentional repetition at successive levels of our curriculum. Of course, ours is a solitary data point, and as such is weak evidence for this assertions. However, we hope that reports of our success will inspire others to mimic our efforts, and provide supportive evidence of their own.

## 6 References

- Ben-Ari, M. (1998): Constructivism in Computer Science Education, *Proceedings of SIGCSE '98*, Atlanta, GA, USA, 257-261, ACM Press.
- Computing Curricula 1991*. The Joint Task Force on Computing Curricula, IEEE Computer Society, Association for Computing Machinery, Available at: <http://www.aqcm.org/education/curr91/homepage.html> Accessed 1 Sept. 2003.
- Computing Curricula 2001*. The Joint Task Force on Computing Curricula, IEEE Computer Society, Association for Computing Machinery, Available at: <http://www.computer.org/education/cc2001/final/index.html> Accessed 1 Sept. 2003.
- Dale, N. and J. Lewis (2002): *Computer Science Illuminated*, Boston, MA, USA, Jones and Bartlett.
- Decker, R. and S. Hirshfield (1990): *The Analytical Engine*, Boston, MA, USA, PWS Publishing.
- Dempster, F. (1988): The Spacing Effect: A Case Study in the Failure to Apply the Results of Psychological Research, *American Psychologist*, **43**:627-634.
- Johnson, D., R. Johnson, and K. Smith (1998): *Active Learning: Cooperation in the College Classroom*, Edina, MN, USA, Interaction Book Company.
- Meyer, R. M. (2003): *Explorations in Computer Science*, Boston, MA, USA, Jones and Bartlett.
- Powers, D. and K. Powers (2000): Constructivist Implications of Preconceptions in Computing, *Proceedings of ISECON '00*, Philadelphia, PA, USA. Available on CD-ROM and at: <http://colton.byuh.edu/isecon/xref/index.html>. Accessed 1 Sept. 2003.
- Silberman, Mel (1996): *Active Learning: 101 Strategies to Teach Any Subject*, Boston, MA, USA, Allyn and Bacon.

Skrien, Dale: *CPU Sim: An Interactive Java-based CPU Simulator for use in Introductory Computer Organization Classes*. Available at: <http://www.cs.colby.edu/~djskrien/CPUSim/>. Accessed: 1 Sept. 2003.