The Architecture Curriculum at UC-Davis

Matthew Farrens

Department of Computer Science
1 Shields Avenue
University of California, Davis
Davis, CA 95616-8562
(farrens@cs.ucdavis.edu, http://arch.cs.ucdavis.edu/~farrens)

Abstract

In this paper an overview of the architecture curriculum at UC-Davis will be presented, with an emphasis on the classes offered through the Computer Science Department. Each particular class will be described briefly, and then some personal opinions about one class in particular will be presented.

1. Architecture in the Computer Science Department

The Computer Science Department at the University of California, Davis supports majors in both Computer Science (through the College of Letters and Sciences) and in Computer Science and Engineering (through the College of Engineering). Our core architecture curriculum consists of 6 classes - 3 at the undergraduate level and 3 at the graduate level (see Table 1). Since Davis is on the quarter system (with each quarter being 10 weeks long) there is a total of 30 weeks of architecture instruction at each level. Each of these classes will be described in more detail in the following subsections.

Table 1. Architecture Classes

Type	Name	Title
UGrad	**ECS50**	*Computer Organization and Machine Dependent Programming*
UGrad	**154A**	*Computer Architecture*
UGrad	**154B**	*Computer Architecture*
Grad	**ECS250A**	*Advanced Computer Architecture*
Grad	**ECS250B**	*High-performance Uniprocessing*
Grad	**ECS250C**	*Parallel Processing*

1.1. ECS50

This is the first architecture class students take, and uses Assembly Language as a tool to convey basic concepts in computer architecture. The students write programs on both CISC and RISC architectures, in order to ensure they are exposed to both styles. Basic architectural concepts such as addressing modes, memory layout, instruction set design, etc. are covered in this course. Interrupts are also covered here, since writing interrupt service routines requires an understanding of the architecture and assembly language of a particular machine. This is designed to be a very challenging class, with homeworks and/or programming assignments due 9 out of the 10 weeks.

1.2. ECS154A

This is something of a transitional class, since students who are majoring in Computer Science (CS) have slightly different degree requirements than those majoring in Computer Science and Engineering (CSE). In particular, the CS students are not required to take any digital design classes, while the CSE students are. Thus, the major topics covered in 154A are digital design, I/O (busses, etc.), and an introduction to the memory hierarchy (caches and virtual memory). The digital design portion of the course can be a bit tricky - the goal is to give the CS students enough information to allow them to understand and work with the material in ECS154B, without wasting the time of the CSE majors who have either already had or will soon take a class exclusively about digital design.

1.3. ECS154B

In 154B, more advanced architecture topics are covered - hardwired and microprogrammed CPU design, an introduction to pipelining and multiprocessors, advanced topics in memory hierarchies, uniprocessor performance analysis under varying program mixes, etc. The digital design tools introduced in 154A are used here to actually design different parts of a real CPU.

1.4. ECS250A

In our program, all graduate students are required to take this class. Thus, it covers a fairly wide range of topics and has to be accessible to students with a wide range of backgrounds. Topics covered in this class include the fundamentals of computer design, instruction set principles, pipelining, an introduction to instruction level parallelism, memory-hierarchy design, and some parallel processing. This class includes a project, which is often difficult to do in a 10-week course (since the more interesting material isn't covered until the second half of the quarter).

1.5. ECS250B

This class focuses on the architecture and design of high performance uniprocessors. In this class the students are exposed to the pioneering work done on supercomputers like the IBM 360/91, the CDC 6600, and the CRAY, with an eye towards how solutions to problems developed then relate to current designs. In addition to studying the classics, we look at instruction level parallelism in detail, and also spend time studying the memory system bottlenecks and how they might be circumvented. Students do projects and present results to the class as a whole, to practice presentation skills.

I try to emphasize to students the reasons for studying the old machines - many (if not most) of the problems confronting designers today were faced by designers of these machines as well (processor/memory speed imbalance, pipelining hazards, etc.). I try to convince the students that they need to study and be familiar with these old, dilapidated ideas because as technology changes, many of the old ideas are rediscovered and become relevant again. For example, a technique used to solve a problem 30 years ago may not translate directly into the modern era, but two old techniques used together in a novel way might.

1.6. ECS250C

This class emphasizes how to do research in and on parallel architectures, from special-purpose machines to commodity servers. The class is systems-oriented, covering a broad range of topics including VLSI, architectures, operating systems, compilers, and applications. The course emphasizes critical reading of current research papers and the final project focuses on written and oral presentation of original results.

2. Architecture in the Electrical and Computer Engineering Department

The ECE program differs substantially from the one in Computer Science. It has more of an emphasis on hardware, and features a two class sequence (*Computer Structure and Assembly Language* and *Introduction to Computer Architecture*), augmented by several specialty classes (*Microcomputer-Based System Design*, *Digital Systems*, etc.) Since I am a member of the Computer Science Department and not the Electrical and Computer Engineering Department, I will not comment further about their program.

3. The Importance of Fundamentals

In many respects the first architecture class the students are exposed to (ECS50) is the most important class of the entire sequence. If handled right it can set the tone not only for the rest of their architecture classes, but also for many of their other classes as well. We believe it is extremely important to introduce and emphasize concepts and fundamentals in this class. Understanding the fundamentals is key to a good solid working knowledge of architecture in general.

When teaching this class, we have three goals for the students:

1. To gain a solid understanding of the fundamentals of Computer Architecture,
2. To learn how much alike all Von Neumann machines are, and
3. To be able to program any machine with a minimum of effort (in essence, to be able to read a manual).

By way of motivation, we explain that even though people rarely get jobs these days writing programs in assembly language, what they will learn in this class will still be of use to them in the future. The analogy I generally use is that you don't have to know how a car works to be able to drive it, but the more you **do** know about a car the better driver you will become. Knowing what a car does and how it works can help you get better gas mileage, help prolong the life of your vehicle, and helps you make smarter decisions when driving (like *why* to avoid the potholes, why you should change your oil regularly, how to avoid accidents, why not to ride the brakes, etc.). Knowledge is power, and my goal is to fill their knowledge gas tanks.

Personally, I am opposed to using any existing commercially available architectures as a vehicle to teach these fundamentals, because all commercial products have features that exist purely because of manufacturing restrictions in place at the time the processor was designed. Unfortunately, students who are seeing this material for the first time are frequently unable to differentiate between fundamental concepts and machine-specific restrictions and idiosyncrasies.

Therefore I use a very clean generic machine as a teaching tool, which allows me to convey what is important and universal about all Von Neumann machines. I try to focus on clarity of presentation and clarity of concepts, because I want the students to build a simple mental model of a machine which will be refined and expanded throughout their education. Using a generic teaching architecture provides a vehicle to convey *concepts* to students clearly and unambiguously.

Once this knowledge has been imparted, I then move on to commercial architectures and show that they contain all the basics elements presented using the generic machine, as well as their own individual warts and bumps. By taking the generic machine and morphing it into various commercial architectures, I am able to explain how the concepts remain the same and show that only the semantics and/or organization changes. As technology constantly leads to changes in implementations and architectures, I believe this solid grounding in why certain features exist is crucial.

I use a smaller CISC-based generic machine as the

teaching architecture instead of a 32-bit load/store architecture for a number of reasons. Because of its sheer size, the 32-bit machine can be very intimidating to students not well versed in the ins and outs of computer architecture. More importantly, such an architecture also has several features (again due to implementational considerations) that the naive student cannot distinguish from fundamental features. Using a load/store machine exclusively, for example, prevents students from learning about and experiencing the various addressing modes that exist in many architectures. Using a nice simple CISC-based generic architecture allows me to convey the underlying principles directly without having them obscured by particular implementations.

Our goal in this class is not to teach the student how to program commercial product X, but rather to enable the student to (with a little work) program **any** commercial product. Many good trade schools exist for those who only want to learn how to write programs for one particular architecture. We feel our responsibility (as a University) is to teach them how to work with *any* architecture, since the market is constantly changing.

3.1. Implementation Details

As a student, I was never satisfied with the answer "Trust me for now, it will become clear later on". Therefore, I try to have an explanation for everything that I present. My goal is to ask the students to "trust me" as infrequently as possible. We begin the quarter by pointing out how fascinating computers actually are - they are nothing more than millions (billions!) of on/off switches, arranged in groups of various sizes, with arbitrary meanings assigned to these groups. The magic is not in the silicon, but in the way humans have assigned meanings to these groups.

We then talk about what is necessary to do a computation (data, instructions), the subgroups within these (integers, characters, floating point), and how many switches (bits) should be allocated to a given group and why. After we get through the different possible number systems, character representations and ways to encode instructions, we then talk about Memory (a linear array of storage cells) and introduce a 3-operand instruction set that uses only direct addressing (the address of the storage cell is right there in the instruction). Doing this motivates the need for some local memory (storage cells that are close and have their own special names, called *registers*). It also provides an excellent mechanism for getting across multiple addressing modes (immediate, direct, register, etc.), and for motivating and explaining the Fetch-Decode-Execute cycle.

Once we have this simple machine, we stop and explain that we have just finished designing a fully functional computer, and that everything for the rest of the quarter is just refinements of this model. For various reasons (performance being the main one) many other features are added, like more addressing modes, more local storage, I/O, ways to interrupt the processor, etc. etc. etc., but this simple machine contains all the basics they need to really understand.

This approach helps the instructor answer the standard student question, "why did they do it this way?" If the job of explaining the fundamentals is done sufficiently, the student will be able to answer this question for him/herself. Or, alternatively, the professor can ask the students this question, and actually expect to get a meaningful answer!

4. Conclusions

The architecture curriculum at UC-Davis focuses on fundamentals and a thorough understanding of the material. We are constantly trying to upgrade and improve our classes, by obtaining new and more powerful design tools, simulators, more effective presentation materials, etc. As our enrollments continue to climb this is becoming more and more of a challenge, but thus far we have managed to succeed. With enough work we expect to be able to continue to improve the program in the future as well.