

LESSONS LEARNED FROM TEACHING COMPUTER ARCHITECTURE TO COMPUTER SCIENCE STUDENTS

Mitchell D. Theys¹ and Patrick A. Troy²

Abstract - Computer science students require more detail about computer architecture than a black box approach can provide. Teaching the appropriate level of detail and assuring that students understand why the subject is taught are nontrivial tasks. In the Computer Science Department at the University of Illinois at Chicago the approach taken is to present the material from the typical three course computer architecture sequence as a two course sequence. In addition, a variety of simulators are utilized to strengthen the material and help control the topic flow. The simulators used include a programmable logic array software package, a MIPS assembly simulator, and a locally created control code simulator. Teaching the two course sequence has proven challenging. This paper presents lessons learned concerning: (1) the level of coverage required; (2) the simulators used, (3) how to maintain topic flow; and (4) future plans for improving the sequence.

Index Terms – computer architecture, curriculum, simulators, course content, computer science

INTRODUCTION

To effectively make use of present day computer systems computer science students require more detail about computer architecture than a black box approach can provide. Hiding details about concepts such as memory hierarchies, pipelines, and datapaths does not create students that can effectively use the increasingly more complex computers that are currently being produced. The level of detail required by a computer science student is not the same as a computer engineer, and the typical three course sequence (digital logic, computer architecture, and assembly programming) only further increases the number of required courses.

A two course sequence is used within the Computer Science Department at the University of Illinois at Chicago. The topics covered within the two course sequence is the same as within the traditional three course sequence, but the level of detail presented and topic emphasis is different. In addition, software simulators are used to assist the students in their learning of these mostly hardware concepts. Currently a programmable logic simulation package is used to teach digital logic circuit creation, testing, and programming of programmable logic arrays; a MIPS assembly simulator assists in presenting assembly programming and how common programming constructs are

implemented in the architecture; and a locally created simulator to showcase control code programming and how current processors control the datapath hardware. Exposing the students to the simulators allows them to understand and experiment with several low level computer architecture concepts without having to hook up wires, use breadboards, understand LEDs, and other concepts that are not critical to the computer scientist's understanding of computer architecture.

The remainder of the paper is broken down as follows. A brief overview of the current implementation of the two course sequence is provided in "Course Overview." The methods employed to maintain a good flow when moving between the simulators are presented in "Content Flow." The decisions made with respect to the type of simulators chosen are highlighted in "Simulators." The "Lessons Learned" section presents lessons learned from the implementation of the two course sequence. The paper concludes with the "Overview and Future Work" section that recaps the content presented and gives a brief overview of some of the future work on improving the two course computer architecture sequence.

COURSE OVERVIEW

The standard three course sequence in computer architecture consists of three credit hour courses on digital logic, assembly programming, and computer architecture. The digital logic course covers the basics of combinational logic design, sequential logic design, pipelines, and K-maps. The second course covers assembly programming and how higher level languages are represented in assembly. The computer architecture course then finishes the sequence and presents material on how to build a datapath, control a datapath, memory hierarchy, and I/O basics. The topics mentioned are not meant to be exhaustive, but are some of the core topics from the courses. Occasionally a laboratory course associated with one or more of the courses in the sequence is included. With the labs, the sequence is becoming something that a typical computer engineer would take during their undergraduate program.

We have all seen outcomes for courses that state "a student should be able to ..." The question becomes how to present the material in such a way so that the students can meet the outcomes, but also understand and build upon the material they have learned. Discussing the architecture of an Intel 80386 processor in excruciating detail might fulfill the

¹ Mitchell D. Theys, University of Illinois at Chicago, Department of Computer Science, mtheys@uic.edu

² Patrick A. Troy, University of Illinois at Chicago, Department of Computer Science, troy@cs.uic.edu

requirement for the student learning about computer architecture, but would the student understand the pieces of the 80386 that constitute the datapath? Or do they understand which pieces of the 80386 memory interface are available in all processors, and which were added to assist the users? Focusing on a single processor might appear practical, because the students learn something that seems realistic, but the students might not understand the basics.

The approach taken in the Computer Science Department at UIC is to break the topics into two four credit hour courses that cover the same amount of material as the typical three-course sequence mentioned earlier. The first course concentrates on the basics and upon completion the students will have created a simple 8bit datapath. The first course covers binary arithmetic and number systems, digital logic, memory design, and datapath design. The goal in the first course is to have the students create a simple 8bit datapath from scratch. The second course presents the remainder of topics, assembly programming, controlling the datapath (both hardwired and control code approaches), the memory hierarchy (caches), I/O, and how the high level languages constructs are actually performed through assembly code. The second course uses a datapath closely resembling the one designed in the first course to provide an easier transition for the students. The assembly programming is taught using MIPS assembly as a practical example.

Through teaching both courses of the sequence during their initial design, and working with the current instructors, many insights about the teaching of computer architecture have been obtained. An example is how to transition from one topic to the next while also maintaining the “big picture” for the students.

A concern is making sure that an appropriate amount of material is covered in the two courses, and that the level of detail is appropriate for an average computer science student. Many of our students have never considered what is inside the box that they routinely use for entertainment, school work, and communicating with friends. The details of the architecture utilized can explain performance and cost issues that computer science students should understand.

CONTENT FLOW

A very difficult content decision is where to split the two courses so an appropriate amount of material is in each course. The computer science students do not need to know details about transistors and how individual gates are created using transistors. Skipping these topics allows more room in the first course to include memory design and putting the individual components together to create a datapath. The first course has a well defined flow with no extreme switches. A content decision in the first course is whether to discuss control of the datapath during the first course, which is the next logical topic after building the datapath. The

second course does have some issues with content flow as discussed next.

The second course has been taught two ways. The first approach started with control of the datapath and then moved to assembly. The second approach, which was done during the Spring 2003 semester, started with assembly and moved to control of the datapath during the second half of the semester. These two approaches will be compared and contrasted to determine the most appropriate with respect to student learning. The professors responsible for the courses have found good and bad with both approaches. Taking the lead from the students and determining which way the students perform better can help sway the decision about which approach to use. Both approaches for the second course have little time to concentrate on memory hierarchy. During the past semester the cache design topics were covered in the first course and may move to the first course permanently to balance the number of topics covered.

SIMULATORS

Choosing the most appropriate simulator is a constant challenge. A graphical simulator available only on departmental or university machines forces the students to work on campus. Many students have their own computers, some of which are laptops, and would prefer to work on the project from a variety of locations, e.g., work, home, while commuting, etc. The university has a large percentage of commuter students who are on campus for short periods of time and find it hard to work on projects on campus. The solution to this problem is to have the students install a software package on their own machines.

Expecting the entire class to be able to install the same software package on their personal machines is flawed. The multitude of different operating systems, machine configurations, and student abilities means that it is highly unlikely that everyone will be able to install and use the same commercial software package. The solution is to have the students use the software installed on the departmental or university machines.

The cycle of a particular solution solving one problem and creating at least one more can continue forever. Our solution about the type of simulator to use for the courses is still evolving. What has been decided presently is that the students should have some choice in the simulator they use for the projects. The students that are more comfortable installing and configuring the software at home can do so, and those that are not can use the software that is available on campus. When doing group projects this means that the group must decide on the same software, but this does not appear to be that large of a problem. This solution gives students exposure to both commercially available packages and open source packages and allows the students to choose which package they want to use. The students are also exposed to the variety of differing software solutions that provide the same functionality. Giving the students these

choices free the professor and teaching assistant from being a software expert and answering all of the students' installation and configuration issues. It does; however, force the students to problem solve and determine why their installation is not working as intended or why it would not install as expected. The students have made use of the course message board to post their questions and problems to the other students for solutions. This has proven effective and has solved most of the common problems.

The remainder of the section overviews each of the simulators used in the courses, why they were chosen, and our experiences using them. In most cases, there are a variety of software packages available that can be utilized. As our exposure to these packages increases, our preferred choices may change.

Digital Logic Simulator

The first simulator chosen for this simulator was the software package that was included with the computer architecture book by Mano and Kime [7]. This book currently provides a copy of the student edition of the Xilinx ISE 4.2 [7]. Previous versions of the book provided the student edition of the foundation series software. The two packages are similar, but support different operating systems, and details for each one are slightly different. A third version of the Xilinx software is WebPack [10], a free downloadable version that supports more of the newer operating systems. This version is supposed to also be more closely related to the ISE version that currently comes with the Mano and Kime text. Depending on a consistent software package from the book, or WebPack, was not realistic, and another general solution was sought.

During the Spring 2003 semester we tried a policy of not requiring the use of a particular software package. A freeware package, TKGate [4], is installed on the departmental machines, the students get a copy of the Xilinx ISE software with the book, and they can download the Xilinx WebPack. A sample screenshot of the TkGate software is shown in Figure 1 and screenshots from the variety of Xilinx packages are on the Xilinx web site. The students are informed of these packages and also informed that they can use any package that provides a gate level circuit creation tool, and a waveform simulator. There are several other packages that claim to do this, some are shareware, some have a nominal fee, and some are extremely expensive. By not requiring all students to use the same package, the responsibility of the professor changes to more of a problem solving and debugging role, instead of a software expert. The professor can not have detailed experience with all the packages available. Instead the students become experts on their individual package and can provide other classmates with support through the online discussion boards provided by Blackboard [11]. In this way the students learn how to work independently, understand that there are multiple ways to get the same solution, and gain some expertise with real-world problems. During the

Spring 2003 semester students found and used two additional packages Digital Works [2] and B2spice[1]. These packages will be investigated to see if they are an improvement over TkGate to be incorporated in future semesters.

To better facilitate the variety of software packages, the group projects require that a report be submitted that include screen captures of the circuits and waveforms created. Not testing the circuits individually and requiring the students to provide the documentation, facilitates the variety of packages the students might find. This also lessens the amount of work required of the teaching assistant and professor. The report structure forces the students to understand how to test their components without testing every possible input combination. In addition, the report reinforces the students written communication skills, and their cooperation in groups helps verbal communication. As an added cheating prevention measure the files created for the project are submitted at the end of the semester and if required checks can be made to assure that the files are not recycled from semester to semester. The project is also modified from semester to semester to diminish cheating.

Assembly Programming Simulator

A variety of assembly simulators can be found on the internet. The text chosen for the second course focuses on the MIPS programming language [1]. The MIPS assembly is more simplistic than x86 assembly or other choices. There are also several simulators available for MIPS assembly that runs on a variety of platforms [5][6]. This fits well with our thoughts about encouraging the students to work on their own machines. In addition, versions of the simulators come with the Solaris operating system and are available on the departmental servers for student use.

There are many resources available on the Internet that discuss the MIPS assembly simulators. These resources have been useful in teaching the process of going from the assembly instructions to sequences of 0's and 1's to the control code, discussed in the next subsection. The projects also help teach how the higher level programming language constructs are represented in assembly.

The projects using this simulator concentrate on the students getting an overview of assembly. A single large project of a video game is not realistic. There just isn't time in the current format to handle such a project and also cover the control of the hardware that is also a part of material in the second course. As mentioned earlier, making sure the transition from assembly to the other topics is a concern that is always being monitored.

Control Code Simulator

To teach the students the interaction between the memory, assembly instructions, and control code instructions, a home brewed simulator called MythSim [9] was created several years ago. The simulator uses a memory file that approximates the memory of a processor and a ucode file

that contains the microcode for the processor. The projects normally consist of a specification of assembly instructions and the students are required to create the appropriate ucode file and memory files for testing purposes. A sample screenshot is shown in Figure 2. Testing is typically done by the teaching assistants using special memory files. The simulator allows the students to step through one control word at a time and see what occurs in the hardware. A graphical interface is provided that shows the encoded control word from the textual control code, as well as the individual control and status signals on the datapath, register file, and memory interfaces.

The simulator currently runs on a unix platform and has been ported to linux. The porting allows students with some background using linux to get the source and compile it on their own machine. No support is provided for individuals who wish to attempt compiling it at home, but again the discussion board on Blackboard provides a mechanism for students to get help from other students. A student who has gone through the course is working on a graphical and a command line Java version that will be more portable. This will also fit well with our encouraging students to work on the projects on their own machines. This Java version is now in Beta form and is available on the web [8].

The problem with the simulator is introducing it to the students. Unlike the digital logic simulator discussed earlier, it is difficult to integrate the simulator's use in the homework assignments. Typically the textbook uses a datapath and control signals that are drastically different from the one presented in the simulator. The simulator was utilized in one of the three course sequence on computer architecture and taught building the datapath and the necessary hardware of the simulator. A similar approach is now being tried where the pieces of the project from the first course are the pieces required by this control code simulator. When the students get to use the simulator it will look more familiar because they designed a datapath and memory interface very similar to the one used in the simulator.

LESSONS LEARNED

This section provides lessons learned through the process of creating and teaching the two course computer architecture sequence. The lessons include software issues, topic presentation issues, and using the simulators during the lectures.

The software bundled with a book can change with little or no warning to the professor. When ordering books for the Fall 2002 semester the book used from the previous semester was out of print. In reality, the software package included with the book had been upgraded which caused the ISBN number to change. This led to confusion about which book was going to be used, in addition to having to learn the new version of the software package in a short period of time.

There will always be at least one student who will not have access to a machine that has an operating system

supported by the required software package. By not requiring the use of a single package the professor is freed from having to handle these special cases. The students are then responsible for finding a package that can work with their platform, or using the packages provided on campus. Information about the packages that work on various platforms increases from semester to semester and is passed to the next batch of students taking the course. This memory helps the students utilize the packages that previous students found to be the most usable.

When requiring the students to use software packages providing interactive informational sessions is highly recommended. We have taught the course expecting the students "to play" with the software and learn how to utilize it on their own with very poor results. All the packages also include tutorials that can be used to learn how to use the software, but the students procrastinate and ignore the tutorials. During the Spring 2003 semester an on-line chat session was performed which allowed the students to be using their software package and ask questions. This worked very well and many positive comments have been received from the students who participated. The infrastructure for the chat is provided by the Blackboard system [11]. The chat session is also logged so students who participated can go back and review the answers to the questions asked and students that did not participate can read the transcript and have another resource that may answer their questions. Having the professor or teaching assistant use the software in a lecture hall is not a tutorial for the students. Many students will get little out of such an in class event. In general students need to be performing the steps themselves on their machines to get the most out of the tutorial.

The second course was organized slightly different for the Spring 2003 semester. The first approach continues the flow of topics from the first course and is considered a bottom up approach through the two courses. A disadvantage of this approach is that many students have not had exposure to assembly level programming and have difficulties distinguishing between control level and assembly level programming. The second approach presents assembly language as a refinement from high-level languages, control code programming is then taught as a refinement from the assembly. This second ordering of material allowed the students to better understand the progression from higher level languages to control code programs. It is expected that future semesters will implement this organization of material.

OVERVIEW AND FUTURE WORK

This paper has discussed what has been done currently in the two course computer architecture sequence. There are many ideas that the author's have with respect to the organization and implementation of the two course sequence. A few of these ideas are discussed with the overview of the paper.

The current courses are both four hours of lecture without a lab. Some discussions have considered changing the courses to three credit hours of lecture and a single hour of lab work. The lab could then spend more time teaching the students how to use the simulators utilized for the courses.

The teaching of VHDL or Verilog during the first course is currently being explored. As computer science students learning a programming language seems like an obvious choice. But there is some concern that the topics are already too confusing and trying to teach the programming languages will only add to the students' confusion. The Xilinx software provided with the Mano and Kime textbook [7] does provide mechanisms to compile and simulate these languages and future plans include finding other packages for other platforms and then integrating these languages more into the first course.

Balancing the amount of material to be covered in each course is still ongoing. Some semesters the material covered in the first course progresses well and there is room for discussing memory hierarchy. In other semesters, the amount of time required for the 'base' material does not allow for covering memory hierarchy. Determining the appropriate content to balance the two courses is ongoing.

The two course computer architecture sequence in the Computer Science Department at the University of Illinois at Chicago relies heavily on simulators in its curriculum. This paper has presented insights and lessons learned from the teaching and creation of the sequence. This information can be used to help professors that are considering teaching or modifying their computer architecture curriculum in computer engineering or computer science departments.

REFERENCES

- [1] Spice AD v4 digital simulation, www.beigebag.com/dig_simulations.htm
- [2] Digital Works Resource Centre, www.mecanique.co.uk/digital-works/index.html
- [3] James R. Goodman and Karen Miller, "A Programmer's View of Computer Architecture – With Assembly Language Examples from the MIPS-RISC Architecture," Oxford University Press, 1995.
- [4] Jeffrey P. Hansen, "TKGate User Documentation," <http://www-2.cs.cmu.edu/~hansen/tkgate>
- [5] James Larus, "Assemblers, Linkers, and the SPIM Simulator," *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, 1997.
- [6] James Larus, "SPIM: A MIPS R2000/R3000 Simulator," <http://www.cs.wisc.edu/~larus/spim.html>
- [7] M. Morris Mano and Charles R. Kime, "Logic and Computer Design Fundamentals," Prentice Hall, 2001.
- [8] Mythsim Control Code Simulator, www.mythsim.org
- [9] Jon Solworth, "Computer Architecture," manuscript in progress," 2003
- [10] Xilinx ISE Webpack, http://www.xilinx.com/xlnx/xil_prodcad_landingpage.jsp?title=ISE+Webpack
- [11] David Yaskin and Deborah Everhart, "Blackboard Learning System: Product Overview White Paper," <http://ma/www.blackboard.com/docs/wp/LSR6WP.pdf>

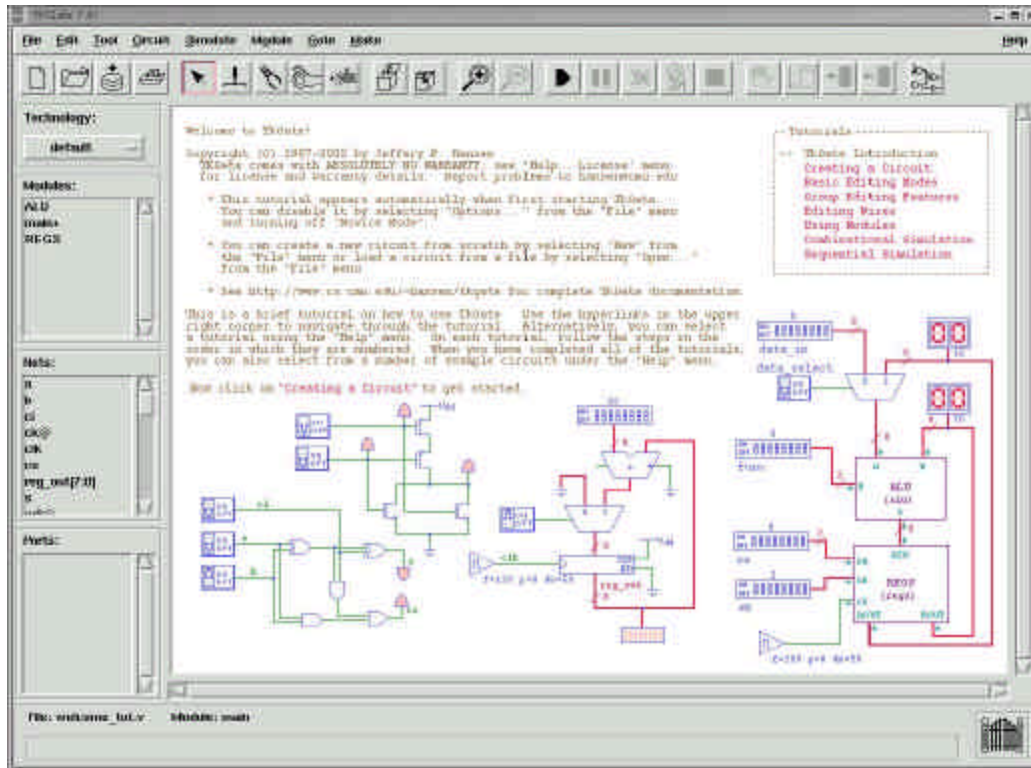


FIGURE 1
A SAMPLE SCREEN SHOT FROM THE TKGATE SOFTWARE PACKAGE

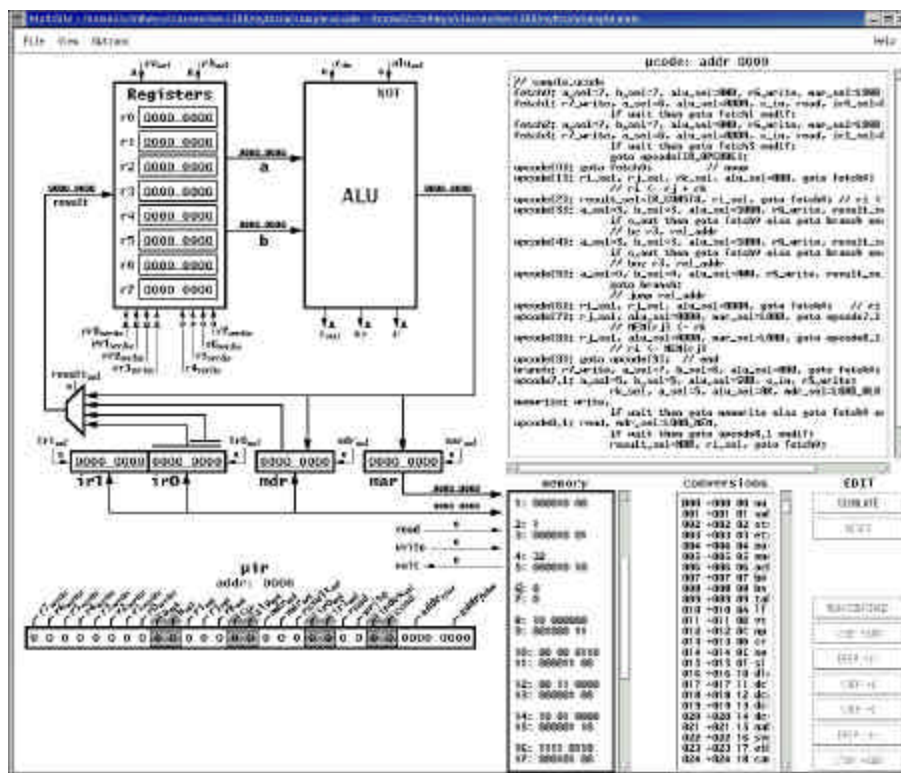


FIGURE 2:
A SAMPLE SCREEN SHOT FROM THE MYTHSIM SIMULATOR SHOWING THE DATAPATH, MICROINSTRUCTION REGISTER, MEMORY FILE AND MICROCODE FILE.