

ON STARTUPS AND TEACHING COMPUTER ARCHITECTURE

Dominique Thiebaut
Department of Computer Science
Smith College
Northampton, MA 01063
thiebaut@cs.smith.edu

INTRODUCTION AND MOTIVATION

The ACM 2001 curriculum [1] states that “[s]tudents need to understand computer architecture in order to structure a program so that it runs more efficiently on a real machine. In selecting a system to use, they should be able to understand the tradeoff among various components, such as CPU clock speed vs. memory size.” Generally, the computer architecture (CA) class in computer science departments is taught in two general flavors: one where the lectures are centered on textbooks such as Stallings[2] or Hennessy and Patterson[3], the other one—usually taught by faculty with more engineering backgrounds—where a digital electronics perspective is presented in lectures and laboratories, in which case textbooks such as Mano[4] are used. In recent years, we have seen new textbooks appear offering VHDL as a conduit for understanding and then programming logic design and microprocessor electronics [5]. In this approach a language such as Verilog is used as a support for understanding and designing simple logic circuits that are easily encapsulated in modules, and in turn integrated into a microprocessor design. Rigo *et al.* [6] describe their experience teaching architecture using a similar approach based on the Architecture Description Language (ADL). Similarly, Gray [7] proposes a whole computer architecture course based on FPGAs.

Rather than discuss the merits of these approaches, we’ll simply note that the second approach is more suited for engineering-rich environments, and would require too long a learning curve for computer science students to be exposed to the full richness of the material.

In this paper we present an experiment we started and that we have found to fully energize the students enrolled in computer architecture. The one-semester course is presented to the students as the setup for a *startup* company, a *venture*, whose final product is a piece of educational software showing graphically the inner working of a computer. As with startup, several key points are emphasized at the beginning:

- Timing is important, and several milestones are set during the semester for design reviews, and presentations.
- Group work is the rule. Each student is responsible for his/her own project, but sharing of code and ideas is encouraged, and each software component is highly modularized with well-defined *Application Programming Interface* (API) for easy interchange of software modules.
- The students are encouraged to influence, even modify the organization of the course and suggest directions, using appropriate channels.

The approach has been used before in capstone entrepreneurship projects in engineering schools, for example, but we found little evidence of it in the context of teaching CA in a computer science department. We have found from experience in the industry, especially in startup companies based on chip design, that before one can have a board with a fully functional chip, simulators are the key to 1) present the concept to potential investors, 2) train new personnel in using the system, 3) design a software development base for the new chip, and 4) to pinpointing target applications. The need to write a simulator for a novel architecture forces the programmer to have a clear, and deep understanding of the architecture his/her simulator must emulate. Thus, organizing a computer architecture class on such principles makes good sense.

COURSE ORGANIZATION

At the beginning of the course, the goals of the class are established: finish the semester with a working graphic simulator of a computer system, with its own CPU and memory hierarchy, that allows users to enter programs written in a native assembly language for the CPU, and executes them, either full speed or in single-step mode. The simulator must have its own documentation, and have some unique features making it different from the others. The course evolves naturally in three different phases.

Phase 1: GUI programming and basic computer organization.

In this phase we introduce Trolltech's Qt GUI software toolkit[8], a software environment for C++, but also ported to Python. Qt is the full-featured, Unicode-aware, cross-platform (Windows, Mac OS, Linux), theme-configurable toolkit that is the basis for the Linux windowing system KDE. It allows the magical act of taking code developed on Windows and compiling it without modification on a Mac or a Linux box. We found Qt to offer a fast learning curve with GUI programming, and the students naturally take to it. In parallel during this phase we introduce the basics of computer organization, the Von Neumann architecture, with registers, ALU, CU, busses, and memory unit. Each weekly assignment is geared to designing a module that incorporates a logic block of the simulator: the processor, the cache memory, the main memory, the control unit, or the ALU, for example. Qt provides widgets that are natural fits for some of these concepts, such as a table widget to represent memory, or the ability to set the background of a rectangular widget to some predefined graphics, onto which other widgets can be located with fixed coordinates. In this case the students can generate a diagram using their favorite drawing package, with boxes, arrows, and other features, then set the resulting image as the background of a frame onto which separate widgets are then "glued." The result is for very pedagogically rich interactive elements, as illustrated in Figures 1 and 2.

Phase 2: On-demand teaching

Once the class reaches the definition of the Instruction Set Architecture (ISA), that is the number of registers, the addressing modes supported, the number of instructions, their distribution in terms of data movement, control, arithmetic and logic, input/output, the students have a good understanding of their task, and manage a software simulator that is taking shape. Typically, they have a module for the semiconductor memory, and one for a cache (direct-mapped or set-associative), and the need to select the ISA parameters naturally brings the student to formulate questions on the choices that must be made to proceed with the programming. Note that our students have taken a semester of assembly language programming before they enter the CA course, and they have a good understanding of the Intel ISA. At this point, the teaching switches from lesson-plan driven to on-demand teaching, with ad-hoc lesson plans driven by student questions and their enthusiasm toward building the next module of their simulator. While, admittedly, this enthusiasm may make them opt for the easiest implementation to get a gratifying quick working design, the established goal of being able to run a simple algorithm such as the computation of the Pascal triangle, with rewards for performance, make the student understand and grasp the notion of design tradeoffs. Homework assignments become action items for the group, and in some cases different assignments are given and students elect one or the other depending on their interest and their expertise. In one case, when the students had implemented the microcode for a few key instructions of the ISA as an assignment, they selected to have two directions for their next assignment, and a small group of students more interested in detail work elected to microcode the whole instruction set, while an other group, more interested in challenges, opted to enhance the memory hierarchy. In both cases the students knew that their code was likely to be used by students from the other group, and they naturally paid attention to clear documentation. The best solution for both assignments, with annotation from the instructor was made available to the whole group after correction.

Phase 3: Advanced concepts

The last month of the semester, the students are deep into programming. This period is reserved for them to add unique features to the simulator, and everybody is encouraged to select something only they will add on their common simulator core. It is a good time at this point to present advanced concepts of computer architecture, such as parallelism (pipelining, dual core, cache coherency), ISA features for multimedia support, virtual memory support, stack-based processors, or reconfigurable architectures. We found that because the students have strong ownership of their project, they express more interest in the advanced topics than in more traditionally taught courses, possibly because they see the parallelism between their software design and the hardware it emulates and can better judge design issues.

SOFTWARE SUPPORT

We found Trolltech's Qt to provide a very steep learning curve for students familiar with C++ or Python (PyQt provides a full API to the Qt toolkit). Several features make this package noteworthy of attention:

1. It supports both a free open-source version and a commercial version.
2. It is a cross-platform toolkit, compiling seamlessly on Windows, Macs, and Linux boxes.
3. It is well documented
4. Its designer tool makes creating windows rich widget an easy and intuitive task
5. It is integrated well with all the C++ compilers available on the different platforms
6. It implements the paradigm of *signals and slots* for communication between modules and widgets that is robust and easy to use.

For the sake of brevity, we explore the last feature listed. The *Signals and Slots* paradigm was introduced by Qt and is a refreshing option to the system of callback functions associated with parent/children widgetry. Under this approach a class implementing a module (say the cache memory) is equipped with special member functions referred to as *signals*, as well as special member functions referred to as *slots*. A module can *emit* signals, and can also have its slots activated by signals generated by other modules. The connection between modules is performed in a central place, very likely the main function.

<pre>// definition of the cache class class cacheClass: public QObject { Q_OBJECT signal: void dataRead(uint, int); public slots: void readRequest(int); }</pre>	<pre>// definition of the CPU class class cpuClass: public QObject { Q_OBJECT signal: readRequestSignal(uint); public slots: void dataReadAvail(uint, int); }</pre>
<pre>// inside the GUI constructor cacheClass* cache = new cacheClass(); cpuClass* cpu = new cpuClass(); connect(cpu, SIGNAL(readRequestSignal(uint)), cache, SLOT(readRequest(uint))); connect(cache, SIGNAL(dataRead(uint, int)), cpu, SLOT(dataReadAvail(uint, int))); ... </pre>	
<pre>// cpu requests data from memory... // at the location stored in the uint address uint address = someFunction(); emit(readRequestSignal(address)); void dataReadAvail(uint address, int data) { // process the memory data stored at location address } </pre>	
<pre>// cache module implementation void readRequest(uint address) { </pre>	

```

if ( cacheSearch( address )==MISS )
    processMiss( address );
else {
    int data = getCachedData( address );
    emit dataRead( address, data );
}
}

```

Table 1. Example of Qt's Signals and Slots paradigm.

Table 1 illustrates the signal and slot concept, and we explore it in details, one row at a time. In the first row, the signals and slots are defined using Qt's new syntax as signal members and public slots members. In the second row showing the GUI constructor, the signals and slots of the modules that talk to each other are connected together. Here the cpu module issues a signal requesting data from the cache, and the cache issues a signal to the processor with the data requested (along with its address). In the third row, the CPU module gets the address of the next piece of data needed, and emits a signal containing the address. Asynchronously, the dataReadAvail() slot will receive a signal from the cache with the data requested. Last row: the readRequest() slot of the cache module is the receiver for the CPU signal. It gets the address as a parameter and performs a search to see if it holds it. In case the data is available (a hit), the data is sent back by emitting the signal dataRead() with as parameters both the address and the data.

Note that neither the cache module nor the cpu module are aware of the other. They simply emit signals and receive signals and work in total ignorance of the entity with which they operate. This results in a clean interface between the modules. When the series of signals and associated slots, along with the information that each pass to the other are well defined, it is easy to maintain modules and insert new ones. For example, if one wanted to make the current cache a level-2 cache and insert in the hierarchy a smaller level-1 cache, all the programming needed (besides the code for the new cache) is the series of connect statements in the constructor of the GUI.

Overall Qt provides quality support to this type of course, economically, pedagogically, especially when interchanging modules between students is an intrinsic part of the course.

COURSE COVERAGE

The topics we covered in this experimental course were the following:

- Boolean logic and implementation of a binary adder with TTL circuits
- Memory hierarchy: cache architecture, cache hierarchy, average access time, replacement policy. Semiconductor memory and error detection and correction.
- Processor design: registers, ALU, and control unit. Algorithms for multiplication and division in the ALU: the booth algorithm for multiplication. Hard-wired versus micro-programmed ALU.
- The IEEE Floating Point standard.
- Pipelining. Collision table. Jump prediction. Superscalar architectures.
- Virtual memory: paging, virtual and physical addresses and caching, TLB.
- CISC and RISC architectures.
- I/O Designs: from the simple parallel port to I/O processors.

TRADEOFFS

The need to cover different topics relating to the Qt toolkit conflicts with the need to cover important material in CA. It is a tread-off, and as instructors we have to justify to ourself that the value added is worth the lack of depth in the study of some areas. We have found that the CA topics that were not covered because of the software nature of the class were balanced by several benefic factors:

- Perhaps the most noticeable is the student's involvement. Driving the course as a venture requires that student play the game of a designer faced with his or her board, and actions must be taken at key points. For example when the design can take different paths represented by potential assignments, the students must voice their choice or suggestions for which path they want to take, and how they want to travel the path. Also, in case where milestones cannot be met, memos must be written by the students to explain their position and suggest options.

- The students quickly understand the ideas of design trade-offs because they are in the position of designers themselves. Should we implement indexed addressing mode, which will force us to add new microinstructions to our current microcode, or is it acceptable for our test assembly language suite to sport convoluted code? Should we implement write-through policy for the cache and take the easy coding route of treating each cache write as a miss, or do we try to gain some performance? And if so, how much performance is gained? They also understand in a clearer fashion the influence architecture parameters draw upon each other, and also how complicated quantifying this influence can be.
- The students quickly understand the advantages of clearly written APIs, good documentation, and efficient programming. The better coders in the group take pride in having their modules used by the other students, and the weaker students benefit from using somebody else's code.
- Students, because of their involvement in the design of a complex system emulating another complex system are more inclined to voice judgment on the merits of solutions and approaches taken by processor and computer manufacturers. Instead of being receptors of information, the students feel they are participants of an exercise that is also played by design teams at computer manufacturers, and as a result have a better appreciation for the ingenuity and complexity of approaches taken by various engineering groups.
- We found that some programming concepts that are important to computer architect researchers but that are normally not covered in a CA course become natural topics to present. A good example is *event-driven simulation*. Ambitious students wanting to show graphically how information flows in the processor, following various paths on the screen, naturally come to want to learn about event-driven simulation, its reliance on event queues, and on timers. Qt provides timers, threading, and locks that integrate nicely for this purpose. Our experience is that the time spent teaching the basics of event-driven simulation gave the students more tools (toys) with which they approached the problem of better understanding of computer engineering concepts in order to better simulate them.
- Ingenuity in using known tools: Macro assemblers are versatile tools, and assembling programs written in a new assembly language can be performed by creating macros for each new mnemonic, and having the macros generate values stored at the appropriate place in memory. Destination addresses of jump and call instructions can be also generated by macros. The students learn how to modify known tools to make them behave in not necessarily their intended purpose, but helping them in reaching their goals.

SOME STUDENT PROJECTS

We present now some of the student projects which best highlight their programming effort. Figure 1 shows the main window of a multi-window project that was specially well implemented. The student, Amanda, made good use of ergonomics to show a large number of details on one window. The left pane contains a stackable menu of options. The center pane shows the processor with its simple architecture, data registers, stack and program counters, internal busses, control and arithmetic/logic units. Note the flag bits associated with the ALU, representing the carry, sign and zero bit. The bottom pane shows the source code whose executable version is currently loaded in memory. This code is read from an assembly file that can be loaded into the simulator, and assembled using the nasm, the open-source netwide assembler. Some basic performance metrics are generated, such as the cache misses and hits. Note also that we used the Qt feature allowing one to use a picture for the background of a widget to show the diagram, then widgets are positioned onto of the picture using absolute positions. Figure 2 shows the approach taken by another student. The last few weeks were used by the students to select various topics we had covered and adding something original to their simulator. This student selected as one of his features the addition of a second level cache which is programmable: the user can select its size, degree of associativity, and line size, and, without stopping the simulator, flush the cache and restart or continue the execution of a program.

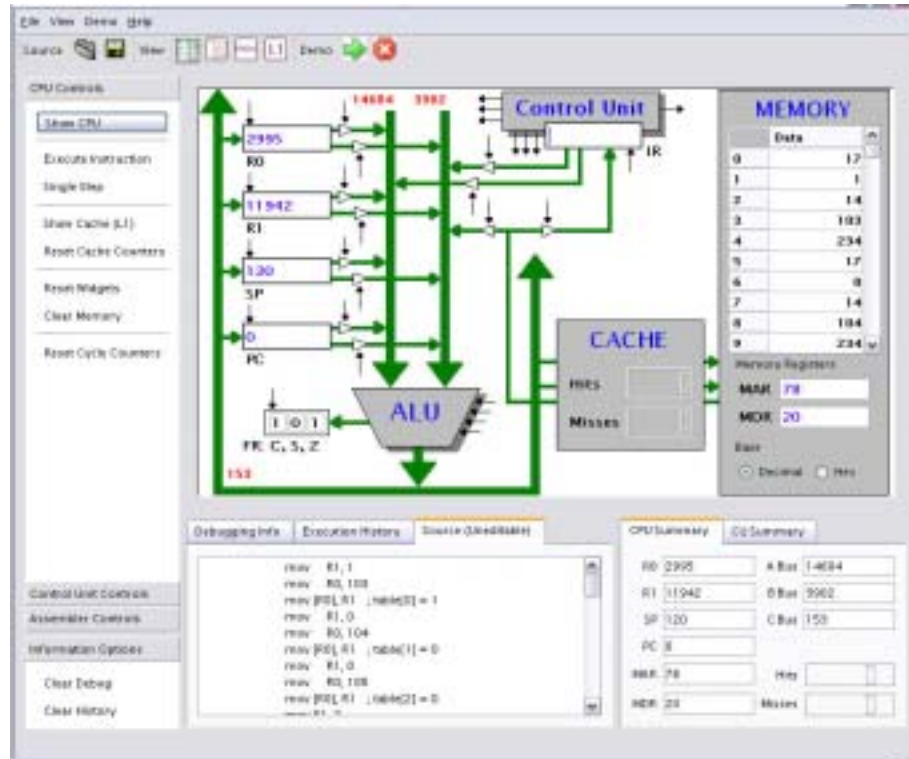


Figure 1: The main window of a project

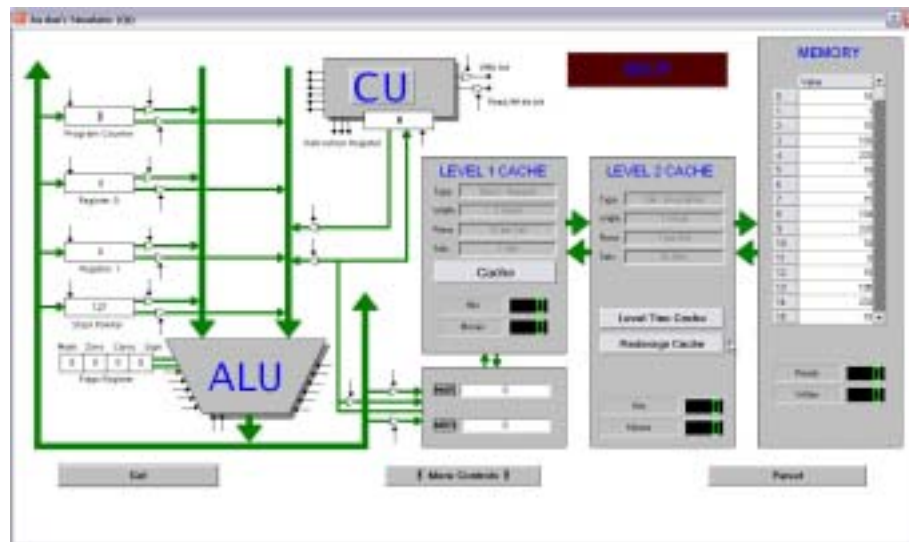


Figure 2: The main window of another project

CONCLUSIONS

While our computer science students are not necessarily all end up programmers in startup ventures, we found that presenting them with an environment where they are put in charge of a project, with strict timing rules, for regular progress reports and presentations, and clear protocols for communication all contribute to motivate them at levels we hadn't reached with the more classic presentation computer architecture. The grading of projects where work is done mostly in a group is still possible as they all have individual projects. Also the students select the unique features they add in the last few weeks in ways that clearly reflect their strengths. Implementing the animation of the booth

algorithm for multiplication is definitely more complicated than adding a level-2 cache, and this can help the grading process.

REFERENCES

- [1] Computing Curricula 2001 Computer Science, IEEE & ACM Joint Task Force on Computing Curricula, December 15, 2001, http://acm.org/education/curric_vols/cc2001.pdf
- [2] Computer Organization and Architecture: Designing for Performance, Williams Stallings, Prentice Hall, 7th ed., 2005.
- [3] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design.*, 4th Ed., Morgan Kaufman, 2006.
- [4] M. Mano, *Logic and Computer Design Fundamentals.*, 3rd Ed., Prentice Hall, 2003.
- [5] S Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design with CD-ROM.*, McGraw-Hill Series in Electrical and Computer Engineering, 2004.
- [6] S. Rigo, M. Juliato, R. Azevedo, G. Araujo, and P. Centoducatte, Teaching Computer Architecture Using an Architecture Description Language, in proc. of ISCA2004.
- [7] J. Gray, Hands-on Computer Architecture –Teaching Processor and Integrated Systems Design with FPGAs, Computer Architecture workshop, ISCA2000, Vancouver, BC.
- [8] Trolltech Inc., www.trolltech.com.