# Introducing Computer Systems from a Programmer's Perspective

**Randal E. Bryant**
**Carnegie Mellon University**
**Computer Science**
**Randy.Bryant@cs.cmu.edu**

**David R. O'Hallaron**
**Carnegie Mellon University**
**Computer Science and**
**Elec. & Comp. Engineering**
**droh@cs.cmu.edu**

## Abstract

The course "Introduction to Computer Systems" at Carnegie Mellon University presents the underlying principles by which programs are executed on a computer. It provides broad coverage of processor operation, compilers, operating systems, and networking. Whereas most systems courses present material from the perspective of one who designs or implements part of the system, our course presents the view visible to application programmers. Students learn that, by understanding aspects of the underlying system, they can make their programs faster and more reliable. This approach provides immediate benefits for all computer science and engineering students and also prepares them for more advanced systems courses. We have taught our course for five semesters with enthusiastic responses by the students, the instructors, and the instructors of subsequent systems courses.

## 1 Introduction

In our experience, we find that our best computer science and engineering students share some common attributes. First, they have a good grasp of the fundamental high-level concepts and abstractions from their programming, data structures, and algorithms courses. They appreciate the power of abstractions and use them whenever possible. Second, and just as important, they understand how these abstractions are implemented on the hardware and software of real computer systems. Our abstractions are not perfect, and when things go wrong (as they often do), good students have the intellectual tools that enable them to determine what is happening at a system level and then correct the problem.

Unfortunately, most computer science and computer engineering curricula (including ours at until recently), never provide students a concentrated and consistent introduction to the fundamental concepts that underly all computer systems. Traditional computer organization and logic design courses cover some of this material, but they focus largely on hardware design. They provide students with little or no understanding of how important software components operate, how application programs use systems, or how system attributes affect the performance and correctness of application programs.

To address this problem, we developed an introductory computer systems course at Carnegie Mellon University in the Fall of 1998, called "Introduction to Computer Systems" (ICS). The course has now been taught for five semesters with two different sets of instructors. Our guiding principles with ICS are to present a more complete view of systems and to do this from a programmer's perspective.

- *Present a more complete view of systems.* The course takes a broader view of systems than traditional computer organization or logic design courses, covering aspects of computer design, operating systems, compilers, and networking. This breadth is crucial for understanding how programs run on real systems.

- *Present systems from a programmer's perspective.* We select material and present it in such a way that it has clear benefit to application programmers. Rather than describing how to design a computer or implement a compiler, the course shows how high-level language programs are mapped onto the machine and executed. Students learn how to use this knowledge to improve program performance and reliability. They also become more effective in program debugging, because they understand the behaviors that can be caused by difficult bugs such as memory referencing errors. In terms of operating systems, the course covers the basics of processes and exceptional control flow. Students learn how application programmers can access these features via calls to the system library. The course also teaches network programming, in order to introduce students to basic concepts of I/O and computer networks, and to give them experience in dealing with concurrency and with client-server computing.

This broader *programmer-centric* approach to systems ensures that the material will be valuable to all students in computer science and engineering. It also clearly delineates the role of this course from subsequent, more builder-centric systems courses. Rather than getting a shallower and more simplified version of material that they will see in more advanced courses, they get a complementary version. By seeing systems from a programmer's perspective, they learn aspects of systems that are covered only indirectly in builder-centric courses.

We have observed a number of benefits arising from the content and style of ICS:

- Students are motivated to learn the material, since they can see how it relates to their needs as programmers.
- It provides good preparation for later systems courses. Having learned the properties of systems visible to programmers, they are better prepared to learn how to implement these systems.
- Since the course is based on the C programming language, which is still the language of choice for system programming, students are better prepared for their upper-level systems courses. A number of ECE courses at our university have made ICS a prerequisite for this reason.
- For non-majors or other students for which this is the only systems course, it exposes them to the most important attributes of computer systems.
- It exposes students to techniques and tools used by system designers, such as debuggers, disassemblers, code profiling, and performance measurements, that can be of great value to application programmers.

The overall experience for the students, the instructors, and the instructors of subsequent courses has been extremely positive. Students find the material motivating and engaging. They report to us later how useful it has been in summer jobs and other courses. Instructors of upper-level systems courses remark how better prepared the students now are to learn about designing and constructing the different components of a computer system.

In the remainder of this paper we describe the topics covered in ICS and how our programmer-centric perspective shapes our presentation. One major strength of our approach is that we have been able to develop homework and laboratory assignments that engage the students' interest and provide them practical skills. They can experience the behavior of real systems running actual programs, rather than relying on pen-and-paper exercises and processor simulators. We describe some of these assignments in conjunction with the topics.

## 2 Logistical Issues

ICS takes one semester, meeting for two lectures and one recitation per week. In the Fall of 2000, the enrollment included 75 CS majors, 56 ECE majors, and 20 others ranging from physics and math to English. Students have already had an introductory programming course in C++ and a data structures course taught in either Java or C++. We also expect students to have had some background in discrete mathematics. No prior experience in OS, hardware, or networking is assumed. Most of the students are Sophomores. In the CS curriculum, ICS has replaced two required courses: one on digital logic design and one on computer architecture. It has been made a prerequisite course for all upper-level systems courses.

All programs in the course are written in the C programming language. C is the language of choice for system developers, because many machine-level features are visible to the programmer, such as bit-level operations, pointer arithmetic, and a non-strict type system. These features, especially pointers, are considered detrimental when teaching introductory programming. They are very useful when teaching computer systems, however, in that many important concepts can be expressed and evaluated using C code. In fact, one important role of ICS is to help students become proficient in C and its system-level features. For example, pointers are much easier to understanding when one understands memory addressing at the assembly code level. The use of bit-level operations for shifting and masking is an important skill that they would otherwise have to learn on their own.

The machine-level programming portion of ICS is highly platform specific. We teach the students machine-level programming for a single combination of machine and operating system. We believe that it is better for students to have a comprehensive experience with one platform rather than less in-depth exposure to multiple platforms. Other aspects of the course, however, have little platform dependence. We have taught the course for two semesters on Compaq Alpha processors running Digital Unix, and for three semesters on Intel Pentium III processors running Linux. We have found both platforms suitable for the course. The RISC vs. CISC issue does not play a major role, since we do not spend any time describing the encoding of instructions or the detailed implementation of the processor. Since Linux runs in "flat, 32-bit" mode, the arcane addressing features of the Intel architecture can be ignored.

## 3 Data Representations

An important concept to convey is that the virtual memory space seen by a programmer is fundamentally an array of bytes. Different data types are formed by grouping bytes into words of different size and interpreting the contents of these bytes in various ways. This is a foreign concept for students used to programming in languages, such as Java, that purposely hide such details from the user.

We cover computer arithmetic, emphasizing the properties of unsigned and two's complement number representations. Unlike a logic design or computer organization course, we do not spend any time describing how arithmetic functions are implemented as logic circuits. Instead we cover features that affect programmers. We consider how numbers are represented and therefore what range of values can be encoded for a given word size. We consider the effect of casting between signed and unsigned numbers. We cover the mathematical properties of arithmetic operations. Students are surprised to learn that the (two's complement) sum or product of two positive numbers can be negative. On the other hand, two's complement arithmetic satisfies ring properties, and hence a compiler can transform multiplication by a constant into a sequence of shifts and adds. We use the bit-level operations of C to demonstrate the principles and applications of Boolean algebra. We cover the IEEE floating point format in terms of how it represents values and the mathematical properties of floating point operations.

Having a solid understanding of computer arithmetic is critical to writing reliable programs. For example, one cannot replace the expression `(x<y)` with `(x-y<0)` due to the possibility of overflow. One cannot even replace it with the expression `(-y<-x)` due to the asymmetric range of negative and positive numbers in the two's complement representation. Arithmetic overflow is a common source of programming errors, yet few other courses covers the properties of computer arithmetic from a programmer's perspective.

Our assignments in this area involve using different bit-level operations to implement arithmetic operations. One challenging example is to implement the C expression `!x` using just the C operators '`~`', '`|`', '`&`', '`^`', '`<<`', '`>>`', and '`+`.'

## 4 Machine-Level Programs

Assembly language provides a portal between a high-level language program and its mapping onto the machine. Being able to read the assembly code generated by a compiler and to understand how it relates to the source code is an essential skill for serious programmers. It allows the programmer to understand the optimization capabilities of the compiler and to find underlying inefficiencies in the code. Programmers seeking to maximize performance of a critical section of code often try different variations of the source code, each time compiling it and examining the generated assembly to get a sense of how efficiently the program will run.

Most computer organization courses show students how to write their own assembly code, often with programming examples that could just as well be written in a higher-level language. Our focus is on being able to read the assembly language generated by a compiler. This involves a different set of skills than writing code by hand. We avoid the tedium of writing and debugging assembly language programs. Pedagogically, dealing with the code generated by a compiler can be somewhat challenging. Hand-generated assembly programs can be written to maximize readability and to introduce different concepts via a sequence of increasingly complex programs. By contrast, one has little control over what kind of code a compiler will generate and what optimizations it will apply to a given code segment. Still, students can accommodate some of these challenges realizing that they are working with real code generated by real compilers.

ICS covers the basic instruction patterns generated for different control constructs, such as conditionals, loops, and switch statements. It demonstrates the typical optimizations performed by compilers such as strength reductions and code motion. We cover the implementation of procedures, including stack allocation, register usage conventions and parameter passing. We cover the way different data structures such as structures, unions, and arrays are allocated and accessed.

An important outcome of ICS is to understand the uses and possible dangers of pointers and pointer arithmetic. Many educators have found that students have difficulty learning to program in C and C++, because they find the effects of faulty pointer code inscrutable and mysterious. Memory referencing errors in C and C++ cause violation of the program abstraction. An assignment operation with an invalid pointer reference to one program object can cause modification of other, logically unrelated program objects. Error messages such as "segmentation fault" or "bus error" are not very helpful to novice programmers. ICS enables students to understand these behaviors and become more effective at debugging.

One of our first teaching assistants, Chris Colohan, developed an especially interesting assignment to give students practice in understanding machine-level code and debugging principles, which we call a "binary bomb." A binary bomb is a program for which the students have only the executable version. It consists of a series of phases. Each phase requires the student to type in a particular string at the keyboard. If the student types in the wrong string, then the bomb explodes by printing "Boom!" and sends an email message to the grading server. Students lose some 0.25 points for every explosion, so there is a real consequence for exploding the bomb. If the student types in the correct string, the phase is "defused" and the bomb sends email to the server with the authenticating string. It then moves on to the next phase. The task for the students is to reverse engineer the program sufficiently to deduce the strings they should supply to it. The bomb phases get progressively harder to defuse. We even include a "secret" unadvertised phase that the students can defuse for extra credit. When all the phases are defused, the bomb is defused, completing the assignment The progress of each student is recorded in real time (anonymized) on the course Web page, so students can track how they are progressing relative to the rest of the class.

The bomb is a beautiful assignment in many ways. For the instructors, it is entirely self grading. For the students, it makes learning machine-level programming feel more like a game than a chore. It also forces students to learn to use a debugger. The *only* way to defuse a bomb is to disassemble it and then use the debugger to explore the program behavior. The bomb lab teaches students about machine language in the context they will most likely encounter in their professional lives: using a debugger to reverse engineer machine code generated by a compiler.

## 5 The Memory System

The memory system is one of the most visible parts of computer system to application programmers. Although virtual memory provides the image of a large, flat address space, features such as caching can have a major effect on program performance. We therefore go into more detail about cache design than about any other hardware component. In addition, having a basic understanding of disk storage and its latency and bandwidth characteristics is important to understanding the motivations for some attributes of virtual memory.

Advanced courses in computer architecture spend considerable time on caches, covering different associative structures, indexing mechanisms, and methods of enhancing the performance during write operations. In ICS we cover only the basic forms of caches. To a first order, the most important performance properties are the block size and the total cache capacity. We use matrix multiplication to demonstrate how different memory accessing patterns lead to different cache hit rates and hence different overall performance. It is interesting to note that Hennessy and Patterson describe how programs can be optimized for cache performance in their graduate text [1, pp. 405–410] but not in their undergraduate textbook [5]. Furthermore, they present these optimizations as a task for the compiler rather than the programmer. In real life, however, programmers commonly rewrite their programs to enhance cache performance. Lebeck [3] has also proposed introducing cache performance measurement into undergraduate programming courses.

In our laboratory for caches, students try to optimize the cache performance of a matrix transposition routine. In the first part, they use a cache simulator to measure the cache hit rate as the performance metric. This provides them with a simple and predictable measure that helps them understand the behavior resulting from different access patterns. This is the only part of the course where we use a simulator of any kind. The second part of the lab has them optimize the measured throughput of their routines running on an actual machine. Obtaining maximum performance requires them to do other optimizations such as loop unrolling and using pointer code. They learn that the code they wrote to maximize the cache hit rate is too complex to run well in practice. Instead, they must write code that finds a balance between CPU and cache performance.

Our presentation of the virtual memory system seeks to give students some understanding of how it works and its characteristics. More detailed coverage would come in an operating systems course. Students should know how it is that the different simultaneous processes can each use an identical range of addresses, sharing some pages but having individual copies of others.

We cover the operation of storage allocators such as the Unix `malloc` and `free` operations. Covering this material serves several purposes. It reinforces the concept that the virtual memory space is just an array of bytes that the program can subdivide into different storage units. It helps students understand the effects of programs containing memory referencing errors such as storage leaks and invalid pointer references. Finally, many application programmers write their own storage allocators optimized toward the needs and characteristics of the application. As a laboratory, we have students write their own `malloc` packages, measuring the space and time performance on a set of hypothetical allocation/deallocation sequences. Writing a storage allocator gives students some feel for the experience of writing systems software. It becomes clear why system programmers write in C where they can circumvent the type system and have control over the run-time system.

## 6 Concurrency and Networking

Most programs that students write in their undergraduate careers read an input file, do some computing, generate an output file, and then quit. However, the programs they will encounter later in their careers will likely have a much richer interaction with the outside world. For example, programs such as network servers run forever, and consist of multiple processes or threads that must interact with each other via shared memory and shared files, and that interact with other computers via network connections.

We introduce the concept of processes and how the system provides the image of simultaneous execution even though only one is executing at any given time. We show them how application programmers can make use of multiple processes via Unix system calls, such as `fork`, `kill`, and `wait`. We discuss exceptional control flow such as interrupts, and provide a programmer's view of interrupts via signal handlers. By comparison, an operating systems course describes how to implement the software that supports process scheduling, context switching, and signalling, but it does not show how these features can be useful in application programs.

We use network programming as a motivator for concurrency, I/O, and client-server computing. We provide an overview of network technology and TCP/IP. For a laboratory assignment, we have students write the user's portion of an Internet "Chat" program. This requires them to concurrently maintain a TCP/IP client to communicate control information with the Chat server, a UDP client to read the stream of chat traffic, and a TCP server to allow other users to query for information about the user. We provide considerable guidance and support on this task, but we have been pleasantly surprised how well students master such arcane features as sockets programming, unbuffered I/O, and `select`.

## 7 Discussion

In order to make room for the considerable material on programming, operating systems, compilers, and networking, we must omit much of the material traditionally found in a computer organization course. We do not cover any logic design or low-level hardware architecture. Our only coverage of CPU architectural concepts such as pipelining is to present an abstract execution model of an out-of-order processor so that students can understand the performance characteristics necessary for program optimization. We do not have students write any software that would have to run in kernel mode, and hence we do not cover device drivers. We believe material such as this can be covered better in more advanced systems courses, when they have gained more experience and can take part in more interesting projects.

In fact, we question why undergraduate computer science programs feel obligated to require *every* student to learn many of the topics covered in computer organization. Why should all students learn logic design, when almost all computing professionals are well removed from such details of the logic technology? Why spend hours on instruction set encoding, when even those who routinely deal with assembly code let assemblers and disassemblers handle such details? Why give a detailed presentation of the operation of a 5-stage RISC pipeline, given that the behavior and code optimization strategies for these machines have little relation to those for a modern, out-of-order processor? Certainly, such material should be presented to students in computer engineering and others who will need to work closely with hardware, but this is only a fraction of the computer science student population.

Patt and Patel [4] have also developed a fundamentally new approach to teaching lower-level aspects of the computer system. Among their goals are some very similar to ours. On the other hand, they present their material from the bottom up, attempting to span from individual transistors to C programming in a single semester. Furthermore, they advocate presenting this material as a first course in computing. We believe in a more top-down approach where students first become familiar with the level of abstraction provided by a high-level language and then learn about the implementation and about some nuances of the underlying implementation. Furthermore, we make no attempt to delve deeply into hardware design.

The current IEEE/ACM effort at devising new standards for computing curricula [6] recognizes that as the computing discipline broadens, the educational priorities must shift to cover new topics while allowing coverage of others to be diminished. In addition, there is a recognition that there should be some definable core that can be expected of any degree in computer science or engineering. The current draft curriculum [2, Appendix A] includes 33 core hours of computer architecture out of a total core allocation of 277. Among these are 10 hours on CPU design. We feel this level of coverage is excessive. On the other hand, compilation does not appear in the core at all. There is nothing listed in the core that shows the connection between a source code program and its machine-level realization. On a positive note, there is a recognition that some coverage of networking must be in the core.

Most people agree that the ability to write programs is a central component of this core. We believe a course in the style of ICS would provide a suitable way to present the computer systems portions of the core. It provides a broad overview of computer systems, combining all aspects of systems into one course. By presenting from a programmer's perspective, it reinforces the sense that the ability to write programs is a key objective for any computer science or engineering program. ICS augments the traditional presentation of programming with a deeper insight into the system that generates and runs these programs. In around 36 hours of lecture time, we cover what we argue represents a better-focussed core for computer systems. It could replace what is currently listed as requiring 59 hours, including all of the core material from architecture and networking, as well as much of the operating systems core.

## References

[1] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, second ed. Morgan-Kaufmann, San Francisco, 1996.

[2] IEEE Computer Society, and ACM. Computing curricula 2001. Draft, Mar. 2000.

[3] Lebeck, A. R. Cache conscious programming in undergraduate computer science. In *SIGCSE* (Mar. 1999), ACM, pp. 247–251.

[4] Patt, Y. N., and Patel, S. J. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. McGraw-Hill, 2000.

[5] Patterson, D. A., and Hennessy, J. L. *Computer Organization and Design: The Hardware/Software Interface*. Morgan-Kaufmann, San Francisco, 1997.

[6] Roberts, E., LeBlanc, R., Shackelford, R., and Denning, P. J. Curriculum 2001: Interim report from the ACM/IEEE-CS task force. In *SIGCSE* (Mar. 1999), ACM, pp. 343–344.